UNIVERSITY OF OSLO
Department of Informatics

Galaxy Portal

Cross-platform app development and dissemination

Master's thesis

Claus Børnich

Spring, 2015

Abstract

This thesis presents the development of an app for mobile devices intended to access the Galaxy bioinformatics service to monitor the status of running analysis jobs.

The Galaxy service is widely used in bioinformatics and uses a web-interface, but the interface design did not anticipate the rapid adoption of mobile technologies, such as smartphones and tablets, and so is difficult to use on a small touchscreen device. A bioinformatics project in Galaxy can often involve several steps where each step depends on the results of previous computations. The analysis processes for such computation jobs can run for a long time, potentially days or weeks. Further progress often depends on the result, and may require new such jobs, creating a chain of analysis processes building upon previous results. The Galaxy Portal app developed in this thesis provides a way for researchers to easily monitor the status of their jobs on a mobile device as a more convenient and efficient way to track progress than returning to a computer at regular intervals.

The app was developed as free and open source software using the GNU General Public License (GPL), using a cross-platform framework, with the source history, binary releases and user guide freely available on the GitHub code repository. A completed version of the app was also distributed on multiple operating system platforms, including the Android and Apple's iOS marketplace, and so is freely available to be installed on an iPhone, iPad or any Android device. The app is currently on version 1.1 with focus on monitoring running jobs, but also provides additional functionality for inspecting results and metadata.

Developing a mobile app using the GPL open source license created legal challenges that were overcome by dual licensing the software and underlying framework.

As part of this thesis a short paper was written and submitted to a scientific journal, and additional authors from the faculty at the university of Oslo and the Galaxy team in the US were invited to participate.

Preface

With my thesis I wanted to develop not just a proof of concept app, but a complete product available for researchers to install and use. Before starting my master I did not have any experience developing mobile apps and so I had to learn all aspects of the process and I have tried to share the key insights of this learning in this thesis.

In order to achieve my goal of creating and distributing an app there were many practical hurdles that had to be overcome, such as purchasing developer licenses for those platforms, creating marketplace screenshots and descriptions, setting up forums for inviting testers, filling out forms to obtain content ratings and so on. I also decided to write a user guide to illustrate how to use the app and what it could do, and in order to use the open source GNU General Public License (GPL) I had to ensure that the source would be available with each release. Using the GPL caused its own problems which was only solved by obtaining a commercial license for the Qt framework the app is built on and dual licensing the app. My supervisor was very supportive and helped me arrange for the University to cover the costs, but this created its own practical problems of transferring license ownership.

Taken together all the many small and large practical steps that had to be taken to get the app out into the wild added a lot of extra work and time that is not easy to account for. However, going through the process, reaching out to potential users and exchanging emails with Qt support and sales has been enlightening, and provided me with a much better insight into what it takes to turn an idea into a software product.

When presented with the idea by my supervisor of publishing in a scientific journal I jumped at the chance, but it did mean adding a second goal to this thesis. The plan that was agreed and followed was for me to write an initial version without any external guidance or input to include as my own work, and then to invite co-authors to contribute to a revised version for submission.

There is a lot of code written for the app and readers are encouraged to investigate the source, report any bugs found, and if so inclined do a fork and develop it further. Thank you for reading.

Acknowledgements

Geir Kjetil Sandve – My supervisor for giving me the freedom to find my own way while offering advise, support and direction when needed. For organising the weekly Master student group meetings at the BMI department so that we could learn from each other and encouraging a friendly and fun learning environment. For participating in and helping to organise the collaboration on the Bioinformatics journal paper. For assisting in getting developer accounts and licenses, and not least for lending me both his iPhone and iPad to enable the release of an iOS version of the app.

Martin Čech - Galaxy Team Member from Pennsylvania State University for testing the app, contributing to the scientific journal paper and providing valuable feedback pointing out bugs, a better way to log in and not least who's encouragement helped motivate further development at an early stage.

Jonas Paulsen and Eivind Hovig – For agreeing to work with me on the scientific journal paper and providing suggestions for both the paper and the app.

Sveinung Gundersen and Morten Johansen – For help getting started with Galaxy when I did not know anything about how to use the API.

Wolfgang Leister and Trenton Schulz – For taking the time to answer my questions about licensing, and to Wolfgang and Nils Damm Christophersen for teaching a very interesting and incredibly useful course on the subject of licensing and FOSS.

Nils Damm Christophersen – For being interested, encouraging and contributing to make the university a friendly place.

Family – Last, but certainly not least. Making it possible for me to do a degree by providing me with invaluable support, understanding and not least assuming both the night and day shift on countless occasions in the loving care of our baby daughter so that I could work on my app, journal and thesis.

Contents

Part I - Background 17

1 Introduction 19

1.1 Goals ... 21

1.2 Overview ... 21

1.2.1 Part I - Background ... 21

1.2.2 Part II - Implementation ... 21

1.2.3 Part III - Dissemination ... 21

1.2.4 Part IV - Discussion ... 22

2 Mobile Technologies 23

2.1 Mobile Applications ... 23

2.2 Mobile Software Platform Market ... 24

2.3 Cross-Platform Deployment ... 27

2.4 App Design Considerations ... 28

3 Licensing 30

3.1 Open Source Software ... 30

3.2 Licensing of the Software Ecosystem ... 31

3.2.1 GPL and LGPL License ... 31

3.2.2 MIT License ... 32

3.2.3 BSD License ... 32

3.2.4 Apache License ... 33

3.2.5 Academic Free License ... 33

4 Mobile Development 34

4.1 Mobile Platform Development ... 35

4.1.1 Android App Development ... 35

4.1.2 iOS App Development ... 39

4.1.3 Windows Phone ... 40

4.2 PhoneGap....41
4.2.1 Background....41
4.2.2 HTML, CSS and JavaScript....41
4.2.3 Cordova Architecture....43
4.2.4 PhoneGap Build....44
4.2.5 Considerations....44
4.2.6 Conclusion Regarding Suitability for Galaxy Portal....45
4.3 Kivy....45
4.3.1 Background....45
4.3.2 Python and Kivy Language....46
4.3.3 Kivy Architecture....47
4.3.4 Kivy Deployment....48
4.3.5 Considerations....49
4.3.6 Conclusion Regarding Suitability for Galaxy Portal....49
4.4 Qt....50
4.4.1 Background....50
4.4.2 C++ and QML....51
4.4.3 Qt Architecture....53
4.4.4 Qt Application Deployment....54
4.4.5 Considerations....55
4.4.6 Conclusion Regarding Suitability for Galaxy Portal....55
5 Galaxy **57**
5.1 Galaxy....57
5.2 Galaxy API....59
5.2 Previous Solutions, Wrappers and Extensions....61
5.2.1 Galaxy Mobile....62
5.2.2 Galaxy BioBlend....62
5.2.3 Genomic Hyperbrowser....63
Part II - Implementation **64**
6 Implementing Galaxy Portal **66**
6.1 Code Development Practices....66

6.1.1 Style....67
6.1.2 Methodology....67
6.1.3 Source Control....68
6.1.4 Working with large third party frameworks....68
6.2 The QML Structure....69
6.3 The RESTful API interface....69
6.3.1 Histories and Jobs....70
6.3.2 Job Status....70
6.3.3 API Interface Interaction....71
6.3.4 JavaScript Object Notation (JSON)....72
6.4 Model-View Design Patterns....74
6.4.1 The Qt Model-View-Delegate Pattern....75
6.4.2 Decoupling Model and View....76
6.4.3 Implementing the Model-View-Delegate Pattern....76
6.5 Navigating Views....78
6.5.1 Show and Hide Components....78
6.5.2 Animated State Transitions....79
6.5.3 QML Loader....81
6.6 Device Screen Scalability....82
6.6.1 Pixel Density....82
6.6.2 Resolution Categories....83
6.6.3 Device Independent Font Size....84
6.7 Iterative Development Process of the Settings....85
6.7.1 Login....86
6.7.2 Touch-Based Paste Functionality....86
6.7.3 Sliders....86
6.8 Event Thread and Background Thread....87
6.8.1 Signals and Slots....88
6.8.2 Background Processing on Mobile Devices....89
Part III - Dissemination **91**
7 Distribution **93**
7.1 Releases....93

7.2 iOS and GPL Conflict....94
7.2.1 App Store Terms and Conditions....95
7.2.2 Dual Licensing....97

8 Writing for a Scientific Journal 100

8.1 The Writing of a Journal Paper....100
8.2 Original Draft....101
8.3 Paper Writing Process....103
8.4 Changes From Collaboration....103
8.5 Final Version....104

Part IV - Discussion 106

9 Final Remarks and Further Work 108

9.1 The Licensing Challenge....108
9.1.1 The Galaxy Portal License Situation....109
9.2 Cross-Platform Development....110
9.2.1 Cross-Platform Testing and Deployment....111
9.2.2 Qt and Cross-Platform Development....112
9.3 Overall Outcome....113
9.4 Future Development....113

Appendix 117

A User Guide....118
B Galaxy Portal Code and Binaries....119
C Journal Papers....122
D Android, iOS and Windows Releases....124
E Collected Gartner Data....125

List of Tables

Table 1: Platform versions, API levels and codenames since API level 16. Note that MR indicates a maintenance release....37
Table 2: The four categories of icon sizes selected according to the pixel density of the screen, where pixel density is pixels per millimetre....83
Table 3: Versions released for Galaxy Portal with dates and platforms supported for each version. iOS was only available to testers outside the team from release 1....94

List of Figures

Figure 1: A Photo of the Galaxy Portal app displaying a list of jobs, with a running job colour coded in green..19
Figure 2: Percentage of time spent online using dedicated apps and mobile web browser according to Flurry Analytics based on data from the US..24
Figure 3: Plotted percentage market share of smartphone sales to end users 2007 to 2013 (2011 data omitted) based on Gartner data...26
Figure 4: The Android SDK Manager is used to update and add tools and packages for every Android version...36
Figure 5: Android virtual device manager showing just a few of the available device definitions for running a virtual device...38
Figure 6: A simplified overview of the Kivy architecture with the intermediate communication layer (Kivy API)separating application and native operating system....................47
Figure 7: C++ backend handles the logic, with a QML presentation layer which is presented in a native style on the target platform..51
Figure 8: Shows the division between the Qt GUI C++ and Qt Quick QML modules, the latter of which is used in the Galaxy Portal app..53
Figure 9: Galaxy web interface as shown in the Chrome browser on a desktop computer.
....58
Figure 10: Job status colour coding with the status on the left and colour on the right. Taken from my Galaxy Portal user guide...71
Figure 11: Part of a job list from Galaxy Portal showing the status for two items that have been "flipped" around by tapping them, one which is a running item in aquamarine......71
Figure 12: List of Galaxy histories on left, and list of job items on the right with the top item flipped to show additional data (data shown is configurable)...............................75
Figure 13: Illustration showing how the delegate encapsulates the model with the data and presents it to the view for rendering...75
Figure 14: Welcome screen when starting Galaxy Portal for the first time....................78
Figure 15: Illustration of children of Column arranged vertically, with children of Rows arranged horizontally. Default state on the left. On the right the x-axis offset is set to the width of the screen for the "historyItems" state to transition the jobs ListView on to the screen. The Welcome component would fill the screen below the ActionBar component when visible and so push the Rows container with the History and Jobs off the screen, but is not normally visible and so not shown for simplicity...80
Figure 16: The settings are all on a single page, but on a mobile device the user can only see a segment of the page at any time as shown here from an Android mobile phone...85
Figure 17: Two Galaxy instances that have been saved. Connecting to an instance is done by simply clicking on it in the list...86
Figure 18: Original attempt to illustrate releasing of the Galaxy Portal app under multiple

licenses. Starting from the bottom the image shows two parallel paths of licensing. One with Qt and my source using GPLv3 for an open source GPL release, and another with Qt and my source using commercial licenses for a commercial release aimed at the iOS market..98

Part I

Background

Chapter 1

Introduction

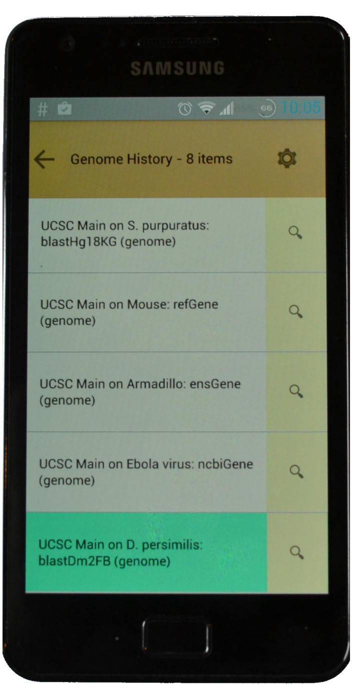


Figure 1: A Photo of the Galaxy Portal app displaying a list of jobs, with a running job colour coded in green.

Bioinformatics is a relatively new interdisciplinary field that tackles problems with analysing large amounts of genomic data. This has created the need for tools accessible to scientists who may not have a background in computer science. The Galaxy platform provides such tools analysing data using a graphical user interface accessed through a web browser, as well as providing storage, and the ability to share and reproduce results.

1. INTRODUCTION

The work in this thesis revolves around the development of an interface to the Galaxy bioinformatics platform aimed at helping researchers monitor the status of long running analysis jobs using a mobile device such as a smartphone. The app allows the user to browse the jobs stored in their Galaxy histories, and displays the status prominently by using colour coding and displaying status and configurable additional information for any item that is selected (selecting an item actually flips it around to show additional data such as status). There are also optional audio push notifications to alert the user of status changes, as the focus of the app is as a status monitoring tool, but the resulting dataset for a job can also be retrieved and inspected through the app.

The app was not only developed, but also released and distributed for free through both the iOS and Android marketplace. In the process of distributing the app issues with using the open source GPL license on the iOS marketplace were encountered, and only resolved by dual-licensing the app.

Having released a working app it was decided to publish the work in a scientific journal by working from an initial draft written by me to a final manuscript developed together with several co-authors recruited in the process.

The many aspects of this thesis has led to a broad focus involving the entire process from initial market research to distribution, and the process of submitting a journal paper for a new software tool.

While the next few of chapters will take a broader view of the app market, licensing and app development we shall return to look closer at the Galaxy system and the purpose of the app interface in chapter five. We will not look at the Galaxy Portal user interface in any detail as I have already written an illustrated user guide (see appendix A) which does exactly that and it is recommended to read through this guide to gain a fundamental understanding of what the app does.

As there is scope for continued work on the app, the thesis was written in a manner that it may also serve to document the process and assist future master students who might continue the work, and parts of this thesis assumes experience with software development, and code examples are used to illustrate parts of the implementation.

A background in biology, or knowledge of bioinformatics, is not necessary as this thesis focuses on software development, licensing and the process of submitting papers for a scientific journal. However, readers with a background in computer science who are interested in a gentle introduction to bioinformatics may want to consider Achuthsankar S Nair's Computational Biology & Bioinformatics: A Gentle Overview (1).

1.1 Goals

- Develop and distribute a software application that can be used by researchers on a range of mobile devices.
  - Research existing technologies to select the framework and tools most suitable for a single person to develop and maintain a cross-platform app.
  - Develop a fully working app.
  - Make the app easily available to researchers.

- Write a scientific paper.
  - Write a scientific manuscript as sole author.
  - Collaborate to write a revised version ready for submission.

1.2 Overview

This thesis spans across a range of subjects such as software development, licensing and submitting a journal paper. To help the reader navigate, the thesis is divided into four parts and a brief overview is presented here.

1.2.1 Part I - Background

In chapter 2 the research on mobile technologies is presented on which I based part of the decisions on what is important in choosing a suitable development framework. Chapter 3 contains a brief introduction to open source licensing as background for the discussion on the licensing issues encountered. Chapter 4 explores mobile development by looking at what is involved in developing for the major mobile platforms and potential strategies for cross-platform development. Chapter 5 takes a look at the Galaxy system the app provides an interface to and describes the application programming interface (API) used.

1.2.2 Part II - Implementation

Chapter 6 presents the work on the actual implementation using examples of the code to illustrate some concepts.

1.2.3 Part III - Dissemination

Chapter 7 looks at the versions released and the obstacle caused by the choice of li-

cense in distributing the app on the iOS app marketplace. Chapter 8 covers the process and experience of writing for a scientific journal.

1.2.4 Part IV - Discussion

Chapter 9 concludes with a discussion on the material covered and recommendations for further work.

Chapter 2

Mobile Technologies

In this chapter we will look at the current software platforms for mobile operating systems used by devices such as smartphones and tablets. This will then form the basis for the discussion of the technologies available for developing apps for these platforms in chapter four.

2.1 Mobile Applications

Increasingly, online access and web browsing is done through mobile devices, such as smartphones and tablets. This trend is widely reported and attributed to the rapid growth of mobile devices worldwide providing affordable and convenient online access[1]. For example, research by the Pew Research Center in June 2013 showed that 56% of adult Americans own a smartphone[2].

Software applications developed for mobile devices are called apps and typically distributed from platform specific marketplaces. While a user can often access websites through a mobile web browser, which itself is an app, the small screen, low precision touch interface, limited processing capability and often limited connection bandwidth makes specialised apps a superior method for accessing online content.

1 Such as reported by the Gartner technology research company; see http://www.gartner.com/newsroom/id/2939217; accessed January 17 2015.

2 See http://boletines.prisadigital.com/PIP_Smartphone_adoption_2013.pdf; accessed January 14 2015.

Figure 2: Percentage of time spent online using dedicated apps and mobile web browser according to Flurry Analytics based on data from the US.

Figure 2 shows graphs based on reports from mobile analytics company Flurry Analytics[3],[4] showing that in the US apps accounted for 86% of all mobile web access in 2014, a 6% increase from the year before. The fact that the primary method for accessing online content on a mobile device is through apps suggests that developing an app will be more successful than trying to adapt the Galaxy website for a mobile web browser format. However, the primary programming language and tools used to develop an app are specific to the operating system of each platform, and so posing a challenge to app developers who want to reach users across device platforms.

2.2 Mobile Software Platform Market

Before we can look at the technologies and choices available when developing for the mobile app market we must first understand how we got where we are today so that we can plan for an app that can survive into the future.

While doing a detailed market analysis is not the intention it is still necessary to con-

3 See http://www.flurry.com/bid/95723/Flurry-Five-Year-Report-It-s-an-App-World-The-Web-Just-Lives-in-It; accessed January 17 2015

4 See http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution; accessed January 17 2015

sider how the situation is developing in order to make informed choices about target platforms and technologies we might use. Aggressive competition has led to rapid innovation and a great range of choice. This is great for consumers, but poses a challenge to developers who wish to develop software that will reach a large audience, and remain relevant and maintainable. This rapid development also means that there are few good research papers investigating the mobile market landscape and those found were often already obsolete. Thus in order to build up an understanding of the mobile platform market we look at published market research. The exact numbers vary from country to country, and also depending on the methodologies used to gather and analyse the data, but the overall trend is always the same and this is our primary concern. We use numbers published by Gartner[5] since this is a widely cited research company that makes summaries of their findings available and reflects the overall worldwide market outlook.

While the idea of transmitting digital information between telephones date back to at least the 70s, as evidenced by a US pattern filed in 1972[6], the smartphone device first came on the market in 1994[7], and the first real mobile operating system, Symbian, only came on the market in 2000[8]. In fact, the leading mobile software platforms of today only date back to 2007 when Apple launched iOS on the iPhone and the Open Handset Alliance led by Google announced Android[9].

To plot a chart showing us how the market share has developed since 2007 and give an idea of the trend for the future figures from Gartner have been collected from 2007 to 2013 and plotted in a chart. The resulting chart shown on the next page gives us a good idea of how dramatically things have changed in just a few years. From Symbian being the established market leader in 2007 with over 77 million smarthpones sold to Android dominating the market in 2013 with 758 million smarthpones sold. Apple's iOS has also continued to grow despite losing market share to Android and are reported to have sold 150 million units in 2013. See appendix E for the Gartner data used presented in tables by year.

5 See http://www.gartner.com; accessed January 17 2015
6 APPARATUS FOR GENERATING AND TRANSMITTING DIGITAL INFORMATION. US Patent 3,812,296; 1974
7 See this Time article for more on the first smartphone http://time.com/3137005/first-smartphone-ibm-simon; accessed January 14 2015
8 See http://www.mobilemag.com/2001/09/25/ericsson-introduces-the-new-r380e; accessed January 14 2015
9 This New York Times article published in 2007 is an interesting look back at the announcement of the Open Handset Alliance and Android http://www.nytimes.com/2007/11/05/technology/05cnd-gphone.html; accessed April 6 2014

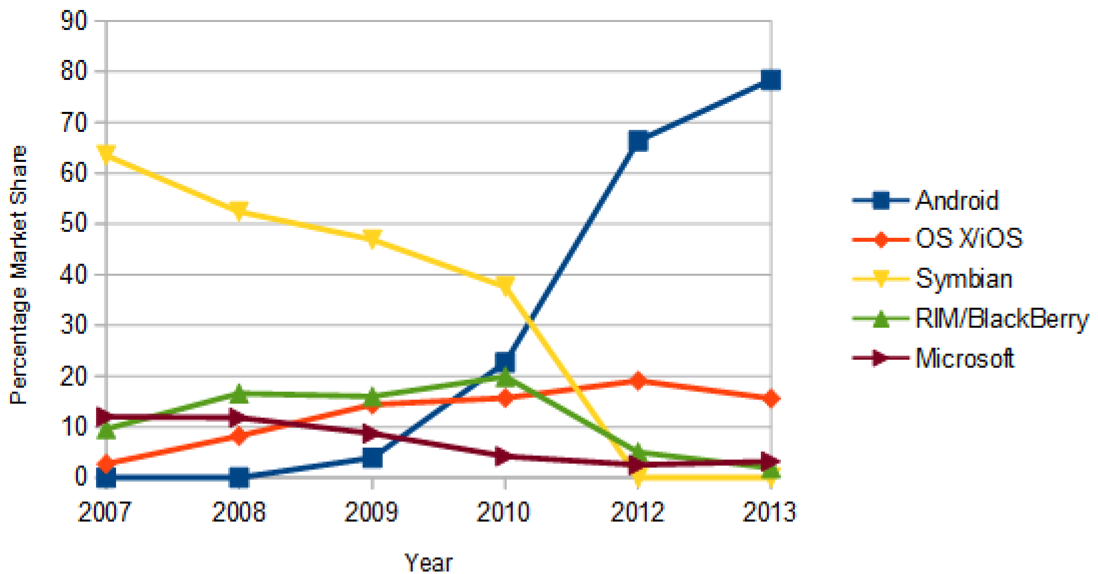


Figure 3: Plotted percentage market share of smartphone sales to end users 2007 to 2013 (2011 data omitted) based on Gartner data.

The tablet market is a similar story with Android now dominating the market and increasing their share of the growing market, followed by iOS[10]. However, according to Gartner sales numbers iOS still had 36% of the tablet market in 2013 with over 70 million tablets sold, while Android climbed from 45.8% in 2012 to 61.9% and over 120 million tablets sold in 2013[11].

These numbers do not tell the whole story of course as the popularity of smartphone and tablet operating systems vary from country to country, and iOS has tended to do better in Europe and America than in Asia where Android has a much bigger market share. Android also has more affordable devices, and so iPhones and iPads are typically concentrated among more affluent groups as reported by the Business Insider in April 2014[12]. Perhaps not surprisingly considering this demographic of users it is also reported widely that iOS users are more likely to download and pay for apps, such as discussed briefly in section 5.5 of "An exploratory study of the adoption of mobile development platforms by software engineers" (2), and so although an iOS app might have a smaller audience it may be a more attentive and interested audience.

10 One example study is this one from 2013 http://marketingland.com/study-android-tablet-users-surpass-ipad-users-by-34m-in-q1-with-tablet-usage-up-282-since-2011-2-47740; accessed April 6 2014
11 See http://www.gartner.com/newsroom/id/2674215; accessed April 13 2014
12 See http://www.businessinsider.com/android-is-for-poor-people-maps-2014-4; accessed January 17 2015

There are also new mobile platforms emerging like Firefox OS[13] by Mozilla which features a HTML5, JavaScript and CSS application layer and web API built on top of the Linux kernel, and Ubuntu Touch[14] which is a mobile version of the Linux based operating system Ubuntu. Even Samsung, one of the largest manufacturers of Android devices, has been developing Tizen[15], an open-source Linux based OS, as an alternative to Android.

This development is important to note as it shows how rapidly the mobile market is moving. At the end of 2010 Symbian was still the market leader[16], but during 2011 both Android and iOS raced past[17]. This illustrates the importance of considering the intended lifetime of an app and strategies for cross-platform deployment.

2.3 Cross-Platform Deployment

The first challenge in writing an app is to select the right technology for the target device platforms and desired functionality. Although Android is currently the most widely distributed platform, iOS has a large and significant market share and as we have seen the current market situation can change quickly and unexpectedly. With this in mind we shall look at a few different strategies to ensure that an app is not locked-in to a single platform.

An app developed for a specific platform is referred to as a "native app", while server-side websites designed for use on a mobile device browser is typically referred to as a "web app"[18].

Native apps are much more common for mobile devices than web apps and perform better with a native user interface that is fully integrated with the platform style and can take full advantage of the platform's capabilities, such as camera, GPS, accelerometer and other device features. The most obvious downside for native apps is the limitation to one specific platform and the high maintenance cost and skill requirement of potentially developing and supporting several versions for different platforms. However, as we shall see later it is possible to use cross-platform tools and frameworks to write an app once and then build it for multiple target platforms. Native apps are primarily distributed through marketplaces, such as Google Play and Amazon Appstore for Android devices, the iTunes App Store for iOS devices and the Windows Store for Windows devices.

13 See https://developer.mozilla.org/en-US/Firefox_OS; accessed April 16 2014
14 See http://www.ubuntu.com/phone; accessed April 16 2014
15 See https://www.tizen.org; accessed January 17 2015
16 See table E2 in appendix E.
17 See http://www.gartner.com/newsroom/id/1924314; accessed April 6 2014
18 For example see this Business Insider article http://www.businessinsider.com/html5-vs-native-apps-for-mobile-2013-6; accessed April 6 2014

We have already seen that most online access is through apps, and from a development perspective a web app can also be an attractive option. A web app is truly cross-platform and available on any computer or device with a compatible browser and access to the server, and maintenance and updates need only be applied to the server. For experienced web developers it is also likely to be faster than developing a native app. The drawbacks of this approach are reduced performance, loss of function when offline, greater security risks and that web apps are not installed but must be navigated to in a user's browser making them less convenient and potentially harder to find. Furthermore, they do not have the same look and feel as native apps making them potentially harder to learn and cannot utilise all the mobile device platform functionality. HTML5 has come a long way in providing the missing functionality for web apps, but has not entirely closed the gap with native apps.

Another popular choice, supported by a number of frameworks, are "Hybrid apps" written using web technologies - typically HTML5, CSS and JavaScript - but run inside a native container that can use the device's browser engine to render the app along with access to the device's features, such as the camera. Different frameworks, such as Phonegap and Titanium, use different approaches to how this is achieved and thus how the app performs, integrates and which device features can be accessed[19].

Hybrid apps balance the strengths and weaknesses mentioned above for native and web apps, and can be seen as a compromise between the two approaches. Specific benefits include a reusable web interface that can be integrated with native code for device specific features and making it easier for experienced web developers to write apps without having to learn the development languages and APIs of each target platform. However, hybrid apps have a higher knowledge requirement than pure web apps as the developer needs knowledge of both web technologies, such as HTML5, and the hybrid framework used to integrate it into a native container. The particular framework used will also introduce its own unique issues, such as adding complexity in debugging, performance issues, limitations in device functionality supported and different ways of rendering the user interface on the various target platforms making the frameworks more or less suited for different types of apps.

2.4 App Design Considerations

This thesis focuses on the process and technology of developing and app rather than design aspects, but as many important design considerations had to be made this section

19 See http://www.appcelerator.com/blog/2012/05/comparing-titanium-and-phonegap; accessed April 8 2014

briefly outlines the main considerations when developing a mobile app based on the platform specific developer documentation mentioned below and best practice guidelines from the App Quality Alliance[20].

Visual design is important on mobile devices as it needs to take into account the low precision touch interface, small form factor, wide range of device formats and the way mobile devices are typically used for quick and urgent tasks. The small screen combined with low precision interface methods also makes data input slow and prone to error. Users also typically use mobile devices while mobile, such as walking or even driving, and so a well designed app should require little effort to use and be easy to navigate with a consistent interface that follows platform standards. The importance of good visual design is reflected in the emphasis placed on visual design in the developer documentation and guidelines for platforms such as iOS[21] and Android[22].

Apps should be designed to minimise their use of power, network, data and system resources such as CPU and memory. Apps also need to be resilient and cache data as the network connection is low bandwidth and high latency, and may switch between using a wireless and cellular connection or shut down the connection when not used, to preserve battery. An app can also be suspended or closed at any time and must handle preserving its state and resuming to a previous state.

Security permissions and privacy settings for an app are also important, but are beyond the scope of this thesis and so not discussed.

20 See http://www.appqualityalliance.org/files/AQuA_best_practices_doc_v2_3_final_june_2013.pdf; accessed April 8 2014
21 See https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/index.html; accessed April 8 2014
22 See https://developer.android.com/design/index.html; accessed April 8 2014

Chapter 3

Licensing

Licensing is an important, complicated and often overlooked aspect of software development, but the choice of licensing can have far reaching consequences that may impact vital aspects such as distribution and ownership. The choice of license had a major impact on the distribution of Galaxy Portal on the iOS App Store and as a result significant amount of work was done investigating and understanding the license challenges. In this chapter we shall look at some of the licenses for software that were encountered and investigated.

3.1 Open Source Software

Open source software is as the name implies software which is distributed with a license that bestows the right for all to access, modify and share the source code and is typically developed publicly by a community of peers. This is the definition of open source promoted by the Open Source Initiative (OSI)[23], one of the two leading non-profit organisations promoting open source software. The other is the Free Software Foundation (FSF) which promotes a free software definition that grants users "the freedom to run, copy, distribute, study, change and improve the software"[24]. The FSF promotes the philosophy of freedom, while the OSI is more concerned with the benefits of open source software development. Although interesting in its own right, we shall not elaborate too much on the history and differences of the OSI and FSF, but awareness of these two camps and their differences can help in understanding the different licenses available and their com-

23 See http://opensource.org/osd; accessed March 8 2015
24 See http://www.gnu.org/philosophy/free-sw.html; March 8 2015

patibility. It is also important to understand that these free and open source licenses rely on copyright laws and should not be confused with software in the public domain which can be freely used without the need for licenses.

The phrase free open source software (FOSS) is often used to include both free and open source software, but in this thesis we simply use open source as this is what is most commonly used when describing all types of FOSS software.

3.2 Licensing of the Software Ecosystem

All the frameworks, the Galaxy system the app is interfacing with and most of the tools and IDEs discussed in this text are open source software, although distributed with a wide range of licenses. However, of the target mobile platforms none are entirely open source, and only Linux is open source among the desktop deployment targets. Certainly a large part of the Android operating system, as developed by the Android Open Source Project (AOSP), is open source, but most of the Google apps and services integrated into most distributions are closed source proprietary products, as are the drivers for the hardware itself. There are several notable AOSP forks – software projects that have branched off the source in their own direction - such as the open source CyanogenMod and proprietary forks such as Amazon's Fire OS, Microsoft's Nokia X platform and SGP Technologies' PrivatOS, as well as Chinese forks.

While it is not the focus of this thesis it is interesting to note that as the Android platform has gained in popularity Google has gradually switched from open source to closed source versions for their apps. These apps are responsible for much of the core smartphone functionality that Android users take for granted such as search, calendar, email, keyboard, camera, photos, maps, app market, movie player, music player and so on.

The iOS operating system has open source components such as the kernel, but the UI is proprietary. The Microsoft Windows Phone is a closed source proprietary operating system. However, all the new mobile platforms emerging that we have mentioned in this work – Tizen, Firefox Os and Ubuntu Touch – use open source licenses.

3.2.1 GPL and LGPL License

To avoid confusion it is useful to know that the FSF was set up by Richard Stallman when developing the GNU operating system (known as GNU/Linux or Linux when distributed with the Linux kernel) and so the FSF published the GNU General Public License (GPL) to ensure that the contributions to the GNU OS would remain free. The GPL re-

quires not only that the source code is made available (as its primary, but not only, requirement), but also that all derivative works use the same license. This is true for any software combined or linked to GPL licensed software and so the GPL is sometimes described as a viral license which "infects" other software forcing it to use the same license to comply legally. Since the GPL uses copyright law to enforce the legal right to access, distribute and modify source code it is often described as being a "copyleft" license.

Other variations of the GPL is the Lesser General Public License (LGPL) which allows linking to software with other licenses and Affero General Public License (AGPL) aimed at software running on a server.

The Qt framework used to develop the Galaxy Portal app is licensed under the GPL version three, typically referred to as GPLv3, and so the Galaxy Portal app itself must also be released under the GPLv3. The LGPL version 3 and 2.1 are also available for the Qt framework, but the GPL was chosen as no additional benefit was perceived by using the LGPL as discussed in chapter seven. When installing Qt the open source edition must be chosen and the terms of either the GPL or LGPL license accepted.

A lot of public commentary on GPL license issues refer to version two of the GPL not GPLv3 which was released by the FSF in June 2007. It is beyond this this thesis to examine the legal differences of these versions, but we can note that version three introduced wording dealing with patents, DRM and hardware restrictions that will in most cases impose additional requirements. However, we can also note that the GPLv3 improved compatibility with other open source licenses such as version 2.0 of the Apache License. The new LGPL version was also changed to refer to the GPLv3, but granting certain additional permissions for using the library with software under different licenses.

3.2.2 MIT License

The QML `JSONListModel` component used in Galaxy Portal to parse JSON data and add it to a QML `ListModel` was based on code released under the MIT license, which is compatible with the GPL license. This license published by the Massachusetts Institute of Technology grants free use, including the ability to sublicense, without restrictions as long as the copyright notice and permission notice is included. Although the final version of the component retains very little of the original code, the license notice is still displayed at the top of the file.

3.2.3 BSD License

A very similar license is the BSD license which was used for the Scrollbar and Flip

components copyrighted to Digia Plc. The new 3-clause license was used, often referred to as the revised BSD license, which removes a clause from the original version which stipulates a requirement to credit a named organisation in any advertising material. This is an important point as the original BSD license is not GPL compatible, while the new revised 3-clause version is. Otherwise the BSD license requires the original copyright notice in the code and stipulates a condition to prevent the organisation or contributors to be used to endorse or promote derived products.

3.2.4 Apache License

PhoneGap uses the Apache license, in particular version 2.0. This is a free software license approved by both the FSF and OSI and widely used. It not only allows modifications, but also allows it to be distributed under a different license as long as the original copyright is used for any unmodified parts. The Apache license is sometimes labelled a "permissive" license together with such licenses as BSD and MIT because it imposes very few requirements, and so is unlikely to run into any conflicts with terms and conditions on any target platforms.

3.2.5 Academic Free License

The Galaxy source code uses the Academic Free License version 3.0[25] which is an OSI approved open source license. However, the license used by the Galaxy system does not have an impact on the choice of license for Galaxy Portal since we are simply requesting data through the Galaxy API, and not building on top of the Galaxy framework or using any of their code.

25 See http://opensource.org/licenses/AFL-3.0; accessed March 10 2015

Chapter 4

Mobile Development

In this chapter we explore some of the current tools, frameworks and programming languages available for app development.

A number of different methods and technologies for developing an app were investigated, but looking at the mobile platform market and considering the limited resources and time available it was decided that a cross-platform approach would be most appropriate.

The target group and their mobile use was not investigated as part of this project, but by aiming for cross-platform support the greatest number of users can be reached while also staying flexible for future changes in the mobile market landscape.

There are a number of interesting technologies for developing cross-platform apps and while it is beyond the scope of this thesis to look at all of them, several interesting alternatives have been investigated in order to make an informed choice.

We shall first provide an overview of some of the technical details, tools and challenges in developing for three of the largest mobile platforms. Then we will examine PhoneGap, Kivy and Qt which were all investigated and tested. Of these Qt and PhoneGap were the most mature and well maintained. In the end Qt was chosen for the sophisticated touch-based user interfaces provided by the QML language, with a wide range of C++ based modules providing rich functionality, backed up by up-to-date documentation, active community and frequent updates with support for the latest devices and technologies. PhoneGap was a viable second choice and if it had been known that the Qt GPL license would prevent distribution of the app on the iOS platform PhoneGap would likely have been chosen due to its less restrictive Apache license.

While the app was intended to be cross-platform it was most frequently tested on the Android platform and so there is more discussion about Android deployment than any other platform. This was partly due to the easy availability of Android devices and the ability to use both Linux and Windows to compile and deploy to these devices, but also because Android was seen as one of the most important platforms to target as elaborated in chapter two. Another reason is that the GPL license used is not compatible with the terms and conditions required to distribute an app on the iOS platform and we will discuss this in detail in chapter seven.

4.1 Mobile Platform Development

Although we shall not attempt to give a complete introduction to mobile app development we shall here try to give a simple overview of some of the particulars of developing for a mobile operating system. We shall focus on iOS and Android which were the most important targets for the Galaxy Portal app, but we shall also take a quick look at the Windows Phone platform which was explored during the later stages of this project.

4.1.1 Android App Development

Developing for the Android platform can be both exciting and exasperating. There is a lot of documentation for just about every framework and the official Android guides for developers are generally very good. The technology is constantly improving and new and better devices are released all the time by competing manufacturers enabling more demanding and powerful apps with better graphics and so on. This development and wealth of choice is of course also a big problem for anyone faced with developing apps on the Android platform. The large user base is fragmented across a bewildering range of devices from different manufacturers and across a wide range of Android operating system versions. Thus when developing an Android app it must be designed to scale according to the capabilities of the device and operating system version. Being compatible with older versions while taking advantage of the latest capabilities of the operating system and devices means more complex code, while restricting the app to newer versions will potentially exclude large portions of the user base. Testing on more than a small subset of the many available devices is in most cases not feasible, although the ability to use virtual devices helps a bit, and so for anything but the most trivial app there is a good chance that there will be device specific issues that will not be discovered.

To deploy an Android app the SDK is required. It can be downloaded from the official

Android developer site[26] for Windows, Linux and Mac in both 32 and 64-bit formats. The SDK install process is just a matter of unpacking the zip file and then running the SDK Manager to update and add packages. By default the packages for the latest API level is installed, but packages for older versions can be downloaded using the SDK Manager.

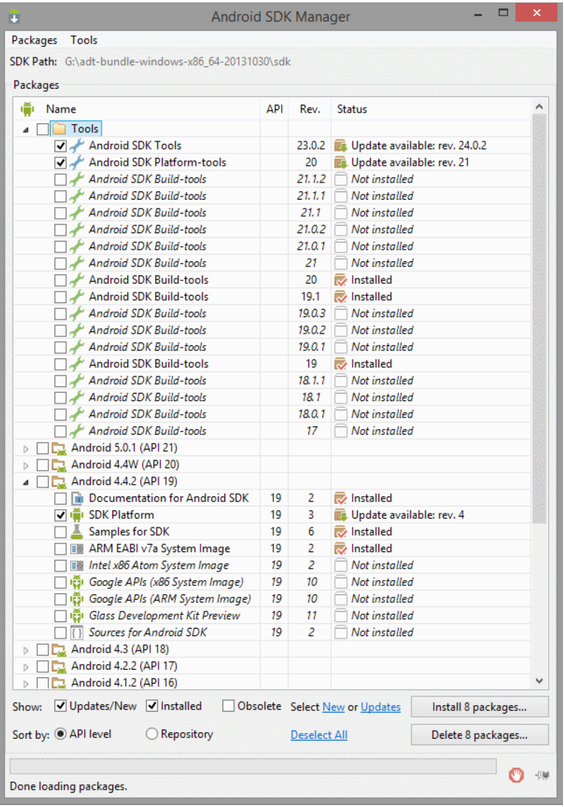


Figure 4: The Android SDK Manager is used to update and add tools and packages for every Android version.

Versions released for a given API level will always be compatible and Android apps can specify the minimum API level they support in the manifest XML file that accompanies every Android app. For example, the Galaxy Portal app has API level 16 as a minimum requirement. Although confusingly the manifest file setting for the API level is "minSDK-Version", the API level is not the same as the Android version and the same API level may correspond to several versions (although most new versions use a new API level). Android also uses codenames, but as a single codename can refer to several versions and API levels they seem to be more for marketing purposes than any useful version tracking.

26 See http://developer.android.com/sdk/index.html; accessed April 9 2014

Version	API Level	Codename
Android 5.0	21	Lollipop
Android 4.4W	20	KitKat Watch
Android 4.4	19	KitKat
Android 4.3	18	Jelly Bean (MR2)
Android 4.2, 4.2.2	17	Jelly Bean (MR1)
Android 4.1, 4.1.1	16	Jelly Bean

Table 1: Platform versions, API levels and codenames since API level 16. Note that MR indicates a maintenance release.

The SDK tools are used to compile code, data and resources into an Android Package (APK) archive file that contains all the app contents and which is used to install the app. Among the contents of the APK file is the certificate, resources, compiled code, raw asset files and the manifest file. The manifest uses the XML markup language to specify important information about the app such as version, names, required permissions, supported screens, and meta data and intent filters with actions and data available to the app.

During development the app can be easily installed on the device using the SDK command line utility Android Debug Bridge (adb) which also makes it possible to debug both virtual and connected devices[27]. Note that to install apps this way on a physical device debugger mode must be enabled and the option for turning this on in newer versions of Android is hidden, but can activated by tapping the build number in the about menu seven times. The app must also be digitally signed to identify the author. This is done during the build process, with release builds using a self-signed certificate stored in a password protected binary file called a keystore.

Once installed the app is available like any other app. Android apps are interpreted into machine code at runtime using the Dalvik virtual machine interpreter which is a variant of the Java virtual machine optimised for mobile devices. In the 4.4 release of Android (API level 19) a new Android Runtime (ART) that aims to improve Android app performance is included as a developer option. ART compiles apps into native code when installed, rather than at runtime, which makes them run faster and more responsive with less drain on the CPU and battery.

In addition to the SDK a Java Development Kit (JDK) must be installed. The standard

27 For the official description for the adb see http://developer.android.com/tools/help/adb.html; accessed April 9 2014

edition of the JDK is open source and either the Oracle[28] version or the OpenJDK[29] version can be used.

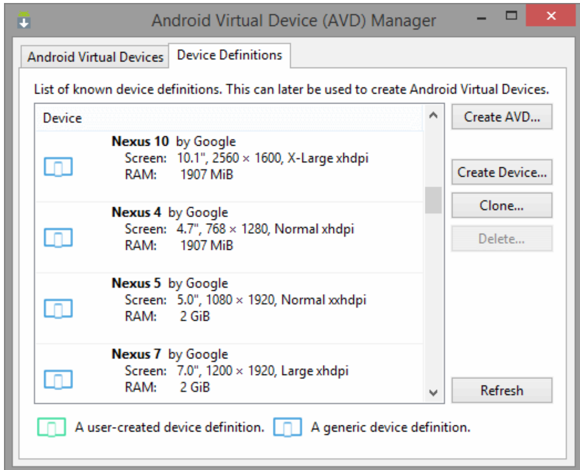


Figure 5: Android virtual device manager showing just a few of the available device definitions for running a virtual device.

The Android Virtual Device Manager is also a part of the SDK and with the system images downloaded with the SDK Manager can be used to configure and run emulators for Android devices. Device definitions can be added with different settings for screen size, resolution, pixel density, input methods, available RAM and whether the device has GPS, Accelerometer, Camera (front or rear) and so on. A list of known device definition allows easily setting up virtual devices for various common configurations and google Nexus devices. The device definitions can be used to create a virtual device for which the platform version, API level and CPU/ABI (e.g. ARM or Intel x86), internal storage and other settings can be selected as long as the appropriate system image packages have been downloaded.

While the APK can be installed on Android devices with debugger mode enabled the app must be uploaded to the Google Play distribution system using the cloud based Google Developer Console to be installed normally. To gain access a developer must sign in with a Google account, agree to the Google Play Developer distribution agreement and pay a $25 registration fee. This gives access to a browser based console where signed APK files can be uploaded and published for alpha testing, beta testing or production. During alpha and beta testing the app is only available to testers who have opted in by

28 See http://www.oracle.com/technetwork/java/javase/overview/index.html; accessed April 12 2014
29 See http://openjdk.java.net; accessed April 12 2014

following a generated URL manually provided by the app developer. Updating the version distributed is as easy as uploading a new APK with a higher version number, and the developer console provides a simple interface to examine app information and basic usage statistics such as the Android version the app is installed on in percentages and crash reports.

4.1.2 iOS App Development

The iOS platform is much less fragmented than Android and devices running older versions tend to be discontinued fairly quickly making support and testing easier as there are fewer devices and versions to worry about. However, the requirements are a lot stricter for releasing apps on the Apple store. Unlike for the Android platform there is a strict review process which an app must pass before it can go into a beta-test available to anyone not registered as part of the development team. The terms and conditions to which an app must conform to pass the review process consist of close to two-hundred points spread across twenty-nine categories[30]. The Galaxy Portal app was submitted and approved by Apple, but the process took eight days in contrast to on the Android platform where it was not reviewed and available a few hours after being submitted.

According to the conditions an app must do something useful, unique or provide lasting entertainment and must be well designed according to their guidelines. There are also many restrictions on the content an app may contain on matters such as religion and violence, and less obvious restrictions such as not mentioning competing mobile platforms. While these conditions certainly made it harder to get the app into early testing on the iOS platform the real problem turned out to be that some of the terms block the use of such open-source licenses as the GPL and LGPL. This legal hurdle turned out to be a bigger problem than the technical challenges and we shall elaborate on this in later chapters.

Unlike for Android, a Mac computer with a recent operating system and the latest version of Xcode installed is required to deploy an iOS app. The iOS SDK is also required and contains the necessary tools and framework for building an iOS app. A device must be connected to actually build a release version of the iOS app and for Xcode to accept this device as a target it must first be registered in the online Apple Developer member centre.

With the iOS SDK installed an iOS simulator also becomes available from Xcode which can be used to run different versions of iPad and iPhone devices to see how the app will

30 See the App Store Review Guidelines: https://developer.apple.com/app-store/review/guidelines; accessed January 25 2015

behave with different versions of the iOS and on different screen sizes.

Similar to the Android manifest, an XML Info.plist configuration file is used by iOS. It is used to set product names, versions, icons and other information relevant to the app such as running in background mode as we shall look at in chapter six. The XML is for the most part a simple series of key-string pairs that set various values for the named keys.

To deploy an iOS app from Xcode it must be assigned a unique bundle identifier (typically based on a URI connected with the project together with the app name) and have a version and build set. It must also be signed with a developer ID certificate and assigned to a team which downloads and embeds the required "provisioning profile" from the member centre after logging in. In this case "the team" was just a single developer enrolled in the iOS developer program which costs $99 per year. We shall not go through all the settings here, but note that the deployment target was set to version 5.1 which was released in 2012 and so supports pretty much all iOS devices except the first iPhone and its successor the iPhone 3G.

The web-based iTunes Connect tools are used to manage the app for testing and distribution. Once the app is compiled, Xcode is used to validate the app and then create an IPA archive file (this is jut a regular ZIP file with a specific file and folder structure). This process will abort and report errors if there are missing icon formats for any supported devices or the Info.plist configuration is incorrect – such as failing to set the "LSRequiresIPhoneOS" key to "YES" to indicate that it is an iOS only app (deploying to Mac OS X requires a separate developer membership). Once all errors are resolved it becomes possible to submit the archived app to iTunes Connect.

In iTunes Connect the app can be made available for up to 25 internal testers registered as part of the team developing the app - for up to thirty days. To progress to external testing, with a limit of 1000 testers, it must first pass the App Store review guidelines. As we shall see in chapter seven, the GPL open source license is not compatible with the terms and conditions, so a commercial license had to be purchased and a new version created specifically for the iOS platform.

4.1.3 Windows Phone

Windows Phone was only fully supported in the very latest 5.4 version of Qt that was released in December 2014 and as we have seen is much less popular than either Android or iOS. For these reasons, and due to the lack of an actual Windows Phone to work with, a Windows Phone app was never released. However, once it became supported in Qt the process was explored as briefly outlined here.

The Windows Phone app must be built on a Windows system with Microsoft's Visual Studio developer tools installed together with the Windows Phone SDK. The developer tools include a Windows Phone emulator for testing. Windows Phone apps were distributed using the XAP archive file format which contained a manifest file and the binary dynamic link libraries (DLLs) that make up the app. However, this is replaced with the release of Windows Phone 8.1 by the APPX file format used for Windows 8 Metro applications.

To distribute the app on the Windows Phone Store a developer account is required. An account for individuals and students can be obtained with a Microsoft account and 113 NOK fee.

4.2 PhoneGap

PhoneGap[31] is a free, open-source hybrid web app framework that can be used to wrap HTML, CSS and JavaScript. PhoneGap dates back to 2008 and has supported a number of platforms over the years and currently lists nine supported platforms including Android, iOS, Ubuntu, Windows Phone 8 and even the new Tizen platform.

4.2.1 Background

PhoneGap was developed by Nitobi, but was acquired by Adobe in 2011, and to prevent issues with trademark the PhoneGap code was released to the Apache Software Foundation with an Apache license under the name Apache Cordova[32]. This means that PhoneGap is a distribution of Apache Cordova, and although Adobe may incorporate additional tools in the PhoneGap distribution there is for most purposes no difference whether the PhoneGap or Cordova package is downloaded and installed[33].

Even though the PhoneGap distribution of Cordova is used it is still referred to as Cordova in the PhoneGap documentation and the command line interface command is still "cordova" and so we shall also refer to it as Cordova, but note that when Cordova is referred to in this section it implies the PhoneGap implementation of Cordova.

4.2.2 HTML, CSS and JavaScript

31 See http://phonegap.com; accessed May 2 2014
32 See https://cordova.apache.org; accessed May 2 2014
33 See this PhoneGap blog for more background on the difference between PhoneGap and Cordova: http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name; accessed May 2 2014

Cordova apps are written using web technologies, and that usually means HTML, CSS and JavaScript. This allows cross-platform development without having to learn each target platform's native development language and for those new to web development there are extensive and very good online references, learning resources and documentation. We offer a brief overview here, but the interested reader can explore the references to learn more.

Most new apps will likely use the fifth revision of the HTML markup language as specified by the World Wide Web Consortium standards organisation (W3C) and usually referred to as HTML5, or the "HTML living standard" as defined by the Web Hypertext Application Technology Working Group (WHATWG) which is a continuously evolving HTML standard. These are effectively two competing standards, but there are very few differences and we shall stick to HTML5 in this text.

HTML5[34] features new and improved markup elements with new input types such as search, email, url, date, color and so on, new form elements and new form attributes, drawing routines, as well as video and audio playback without the need for JavaScript. These extensions to the HTML standard make complex web applications possible and so makes HTML5 suitable for cross-platform development of apps, as discussed by Richard Padley in "HTML5 – bridging the mobile platform gap: mobile technologies in scholarly communication" (3).

CSS3 is the third revision of the cascading style sheet language used to format HTML. CSS3 functionality is divided into modules and introduces sophisticated handling for fonts, shadows, colour gradients, transition effects and transformations that can translate, rotate, scale and skew elements, as well as other forms of styling[35].

JavaScript (JS), an implementation of the ECMAScript standard, is the most popular dynamic programming language used to provide scripting for web pages and is considered an integral part of HTML5. Cordova works well with most JS libraries, such as for example JQuery Mobile[36], AngularJS[37], BackboneJS[38] or Senscha Touch[39]. There are a lot of JavaScript libraries, and these can be used to simplify scripting, as well as provide dynamic views, architecture patterns such as the model-view-controller (MVC) pattern and provide a structure based on best practices through their frameworks.

34 Mark Pilgrim's Dive Into HTML5 is an excellent resource for learning about HTML: http://diveintohtml5.info/index.html; accessed May 3 2014
35 For an introduction to Cascading style sheets: http://www.w3.org/Style/CSS/Overview.en.html; accessed May 3 2014
36 See http://jquerymobile.com; accessed May 3 2014
37 See https://angularjs.org; accessed May 3 2014
38 See http://backbonejs.org; accessed May 3 2014
39 See http://www.sencha.com/products/touch; accessed May 3 2014

4.2.3 Cordova Architecture

Node.js is a server-side platform for running servers implemented in JavaScript and can be be used to install Cordova for the command line. Once installed, Cordova can be executed from the command line to create projects, add and remove target platforms, build Cordova projects and run them on an emulator or connected device.

Cordova uses plugins to provide interfaces to various native components such as battery status, accelerometer, compass, geolocation, camera, vibration, contacts, etc., and the command line interface[40] is used to add these plugins to a project.

Generating a PhoneGap project and adding target platforms is a simple process using the cordova command line utility as shown here.

```
cordova create no.claustest.cordovagalaxyportal 'CordovaGalaxyPortal”
cordova platform add android
```

A basic Cordova project is organised into four directories: "hooks", "platforms", "plugins" and "www", as well as a "config.xml" file which specifies the project id (typically a URL) as well as the name, description and author details.

The "hooks" directory is used to add scripts to execute before and after corresponding Cordova commands to integrate version control and other custom build system functionality.

The "platforms" folder contains the build files for various target platforms such as Android or iOS and is empty until generated with the associated Cordova command, such as "`cordova platform add android`" for Android.

The "plugins" folder holds any installed plugins. These plugins are added directly from their repository using commands such as: "`cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-geolocation.git`".

Finally, the "www" folder contains the main web assets with the HTML, CSS and JavaScript files, as well as resources such as images. An "index.html" web page provides the entry point for the web application and contains the initial HTML and typically links the associated CSS stylesheets and JavaScript script files which are organised in their own sub-folders.

Since the Cordova project is based on HTML, CSS and JavaScript the parts of the app that do not rely on device specific features can be tested in a normal web browser and

40 See http://docs.phonegap.com/en/3.4.0/guide_cli_index.md.html; accessed May 3 2014

web developer tools, like Chrome's developer tools, can be used to analyse and optimise the code. Being able to test on the desktop without having to deploy the app to the target device saves a lot of time and makes debugging much easier.

4.2.4 PhoneGap Build

To build Cordova applications for various target platforms the required SDKs and development tools must be available. Often requiring the build to be done on a specific operating system. To make it easier to create cross-platform mobile applications Adobe PhoneGap provides an online cloud based build service. PhoneGap Build allows a Cordova project to be compiled simultaneously for Android, iOS and Windows Phone 8, although iOS apps cannot be built without a developer certificate. Registering an account is free and there is no need to install anything, but projects must be hosted on the GitHub[41] source repository to be built. GitHub is free for public repositories and so suitable for open source software, but anyone wishing to keep their code private must pay a monthly fee. Once the app is built the resulting packages for the selected target platforms can be downloaded from PhoneGap Build.

4.2.5 Considerations

The cross-platform, open-source and free Eclipse integrated developer environment (IDE)[42] can be used to edit, manage, build, run and debug Cordova projects, but to build, test and deploy the iOS version the Xcode IDE must be used on a machine running OS X. The PhoneGap Build service can of course be used, but does not provide debugging or virtual devices which may be required during development.

One of the big advantages of using PhoneGap is the ability to develop the app as if it was a website, and using HTML5 and sophisticated CSS3 styling allows for a very flexible and professional looking design that can be be responsive to the screen size it is running on and adapt the layout according to the screen resolution. This resulted in very fast iterations of development and testing when developing the Cordova Galaxy Portal prototype. However, the same-origin policy built into browsers prevented using web browsers to test communicating with Galaxy servers as requests cannot be made to another domain. Cross origin communication can be enabled by adding support server side for Cross Origin Resource Sharing (CORS), but that would require running a modified local instance of the Galaxy service. For mobile apps this policy does not apply, but requires adding permission in the iOS configuration file or Android manifest to allow requests to external do-

41 See https://github.com; accessed May 4 2014
42 See https://www.eclipse.org; accessed May 4 2014

mains.

The project was successfully built using both the Eclipse IDE and using the PhoneGap Build service. In both cases the resulting APK was deployed successfully to an Android phone using the Android Debug Bridge, and using CSS3 media queries[43] the layout could be adapted for a range of target screen sizes. The code for a basic login screen "Cordova Galaxy Portal" test project[44] along with its Android platform files are available on GitHub (along with a small "phonegap-workshop" tutorial project[45]).

Note, that to write anything more than the most basic app it would make sense to make use of one of the many good JavaScript frameworks for a better structure for testing, re-usability and easier code maintenance, such as Sencha Touch, AngularJS or BackboneJS.

4.2.6 Conclusion Regarding Suitability for Galaxy Portal

PhoneGap is well supported, uses a permissive Apache 2.0 License and supports a wide range of platforms. Developing the user interface using web technologies, such as HTML, CSS and JavaScript, means that there are a lot of tools and support available, but writing PhoneGap code did not match the speed and elegance of writing in the QML language and took considerably more work to achieve the same quality of appearance and functionality. To achieve the same advanced animations and touch functionality additional JavaScript frameworks would likely have to be used. There were also concerns about reported performance problems potentially leading to a less responsive interface. As such, while PhoneGap was suitable as a framework for the Galaxy Portal app, it was considered only as a backup candidate to the Qt framework.

4.3 Kivy

Another framework investigated was Kivy[46] distributed under a liberal MIT license which means that Kivy can be used in both open source, free and commercial projects without any restrictions or requirements. However, note that there is no commercial support, only community support through user forums and online IRC chat.

4.3.1 Background

Kivy grew out of the PyMT project (4) - a python library for multi-touch applications,

43 See http://www.w3.org/TR/css3-mediaqueries; accessed May 5 2014
44 See https://github.com/Tarostar/CordovaGalaxyPortal; accessed May 10 2014
45 See https://github.com/Tarostar/phonegap-workshop; accessed May 10 2014
46 See http://kivy.org; accessed April 15 2014

but the Kivy framework is not backward compatible due to fundamental changes to improve the design and performance.

According to the paper "Kivy – A Framework for Rapid Creation of Innovative User Interfaces" (5), authored by three of the Kivy core developers, the motivation for Kivy was a lack of cross-platform toolkits for programming Natural User Interfaces (NUI). The two key aspects of the NUI stated by the paper are input handling and output rendering. For input handling new event models were explored focusing on touch capabilities, while for output rendering Kivy uses OpenGL to take advantage of modern graphics hardware and device independence. The paper written in 2011 presents an overview of the architecture and design choices made to make Kivy a powerful tool for developing the next generation user interface.

4.3.2 Python and Kivy Language

Kivy uses Python 2.7 or higher and the entire Python standard library (`stdlib`) is supported. Kivy is a framework on top of Python, so all normal Python features exist, but with extra Kivy functionality added. All other external library imports are also supported (e.g. `NumPy`, `SciPy`).

Python is a high level language which makes it easier to implement applications quickly, but at the cost of execution speed. Python is significantly and often noticeably slower than C or C++ when it comes to tasks such as serious number crunching and drawing sophisticated graphics. Kivy solves this by using Cython to compile critical code such as event dispatching, matrix calculations and graphics rendering down to the C level where the compiler can optimise performance. Custom code can also use Cython to speed execution if necessary. Kivy also uses its own custom drawing compiler to optimise drawing code automatically and shift the load to the GPU.

Although Kivy apps can be written entirely in Python, Kivy also provides its own language stored in files with the “kv” extension and referred to as Kv files. The language is intended to be an easy way to define interfaces and uses CSS style rule definitions and templating. The Kivy language uses indentation similar to that used by Python. Until recently the indentation had to be four spaces, but in the latest versions any number of spaces is allowed as long as it is a multiple of the spaces used on the first indented line. The Kivy language is compared to the Qt QML language in the official Kivy documentation, which we investigate in section 4.4.2.

Writing Kivy apps will be familiar to Python developers and typically consists of writing a collection of classes using the Kivy modules such as `kivy.app` and `kivy.uix.widget` in

python files with the "py" extension. Widgets, such as labels and buttons, can be defined in the Kivy language and Kivy uses a standard naming convention to automatically look for a Kv file named the same as the main App class, but without "App" at the end which is the naming convention for application base class. Alternatively, Kv files can be manually loaded using the Kivy `Builder` class using the `load_file` or `load_string` method.

4.3.3 Kivy Architecture

As a cross-platform framework one of the most important features of Kivy is the intermediate communication layer between the application code and target operating system. This API creates an abstraction layer that separates the application from the operating system allowing it to run on any supported platform. This layer is modular allowing specific parts to be used as needed and this flexibility makes the distribution size smaller. These modules are called core providers and includes such things as displaying text, images and video, and the same approach is also used for input and graphics. This modularity also means that adding new core providers to support new features and input methods can be done by adding new classes which talk to the Kivy application on one side and the target operating system or device input on the other[47].

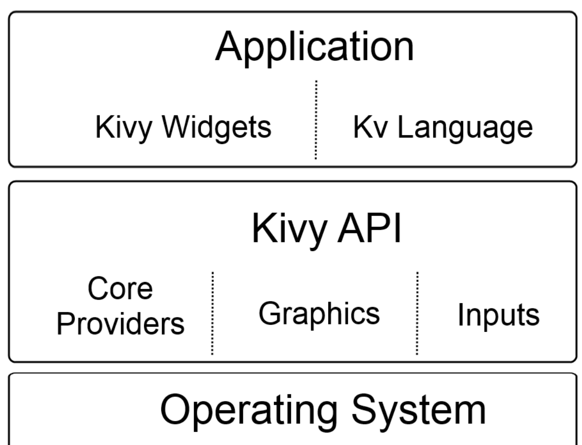


Figure 6: A simplified overview of the Kivy architecture with the intermediate communication layer (Kivy API)separating application and native operating system.

Kivy focuses on touch-based interface, with mouse events translated to simulated touch events, and is based on widgets ordered in layouts. These widgets take the form of text labels, images, input fields and so on which make up the user interface. As mentioned, Kivy is based on the Python coding language and so code is written in Python, or

47 See http://kivy.org/docs/guide/architecture.html; accessed April 15 2014

the Kivy language for widgets.

4.3.4 Kivy Deployment

There is no IDE for Kivy, or plugins to integrate Kivy with existing IDEs, but in the GitHub documentation for Kivy there are some informal instructions for how to configure most IDEs, such as Visual Studio or Eclipse, to work with Kivy. The apps developed to learn the framework were coded using GEdit on Linux and Notepad++ on Windows. These text editors have syntax highlighting for Python (and other languages) and for Notepad++ there is an auto-indent feature and plugins can be downloaded with additional features. Even so there is no project management, auto-complete, debug functionality, user interface designer or other IDE features that provide development support.

Apps can be tested on Linux from the command line with "`kivy <filename.py>`", while on Windows the easiest way to run a Kivy app is to drag the main python file to the "kivy.bat" file in the portable Kivy directory. This sets up the environment for Kivy and starts the Python interpreter.

A Linux platform is required to deploy Android apps, so although the code could be written and tested on Windows or Mac, the Android packages must be created on Linux. In addition to the Android SDK and JDK as described in the section on Android app development, and the Android Native Development Kit (NDK) and Apache ANT build tool mentioned in section 4.4.4 on Qt application deployment, the Jinja2 python module and Python for Android must also be installed to deploy Kivy projects to Android. In particular Python for Android takes care of the APK packaging and contains a small API for bootstrapping the Android app. It is downloaded from GitHub[48] and then has to be built for Kivy using "`./distribute.sh -m "kivy`" which creates the distribution.

On Ubuntu 13.10 a number of additional packages[49] also had to be installed to create the minimal build environment for building Python Kivy apps for Android and the latest version of Cython was also needed. Instructions for setting up packages and the environment variables correctly were not discovered in the Kivy documentation and so required research as well as trial-and-error. Once the Kivy app has been built the familiar APK file was generated for Android and could be installed and run on an Android device using the Android Debug Bridge command line from the Android SDK.

48 See https://github.com/kivy/python-for-android; accessed April 17 2014
49 The packages are: build-essential, patch, git-core, ccache, ant, python-pip, python-dev, ia32-libs, libc6-dev-i386, lib32stdc++6 and lib32z1

4.3.5 Considerations

The biggest hurdle in using Kivy was the large number of third party packages and utilities needed to deploy a project to a platform such as Android, and this was made worse by the lack of well organised and accurate documentation. The available online documentation was not always complete or up-to-date. For example, the programming guide sections on "Best Practices" and "Advanced Graphics" were empty placeholders for the unreleased version 1.9 in April 2014 when the Kivy test app was developed and had still not been updated January 2015, while other parts of the documentation still discuss various older versions of the framework.

While stability can be a good sign for a mature framework it is also of some concern that the last version – 1.8.0 - was released over a year ago in January 2014. However, there are a number of Kivy apps being released and being open-source we can see from its GitHub source repository that it is still being actively developed. At the time the test apps for Kivy were written there was only a single 138 page book available (6) on using the Kivy framework, but since then several more books have been released.

It is important to consider these things as we are looking for a cross-platform framework that will ensure the survival of the app into the future, and while Kivy is an interesting framework that allows rapid development using a high-level language there is also a risk that Kivy will not be developed further and so become obsolete.

A very basic login test app was developed and is available on GitHub as Kivy Galaxy Portal[50] - as well as a simple, but fun Kivy Pong tutorial[51]. Both projects were successfully deployed to Android (using the adb), Windows and Linux, and consisted of only a few files, which in the case of the test app project consisted mainly of a kv language file and three simple python files.

4.3.6 Conclusion Regarding Suitability for Galaxy Portal

Kivy was interesting for using Python and having a different look-and-feel from the user interfaces developed with other frameworks. However, it was severely lacking in support and tools, and with no new releases in over a year confidence was low about the future of the framework. It was therefore not considered a suitable framework for continued development.

50 See https://github.com/Tarostar/KivyGalaxyPortal; accessed April 17 2014
51 See https://github.com/Tarostar/KivyPong; accessed April 17 2014

4.4 Qt

"Qt is a cross-platform application and UI framework for developers using C++ or QML, a CSS & JavaScript like language. Qt Creator is the supporting Qt IDE"[52]. Elaborating on this quote from the Qt project website, Qt is an open source framework that makes it possible to write code once and then compile it for a range of target platforms. The same code can be compiled for both desktop platforms and mobile platforms, and Windows, Mac OS X, Linux, Android and iOS are all supported. Qt is owned by Digia Plc who sells a commercial license, but Qt is also available under both a LPGL 2.1 and GPL 3.0 license which means that the Qt framework and its libraries can be used for free as long as the license conditions are met.

Online installers which will download content based on selected components and complete offline installers are available for a Linux, Mac and Windows. Both binaries and source packages for the framework libraries, as well as the Qt Creator IDE can be easily downloaded and installed using the installers, or built from the source.

4.4.1 Background

Qt has a long and interesting history under several companies and licenses, but that is beyond the scope of this paper. Suffice to say that the framework was released in 1995 by Trolltech, although development stretches back to 1991 and the framework has gone through five major versions in that time, as talked about in the book *C++ GUI programming with Qt 4* (7).

In the last few years Qt has shifted from being primarily a framework for cross-platform desktop applications to also include mobile platforms. With Qt version 5.0 QML and JavaScript became the primary means for developing the user interface, with C++ still powering the backend. QML is a declarative language that resembles and can be used together with JavaScript, and allows rapid development of modern user interfaces. With version 5.2 in December 2013 Android and iOS was officially supported and in December 2014 Windows Phone and WinRT support was added.

52 See http://www.qt.io; accessed March 25 2015

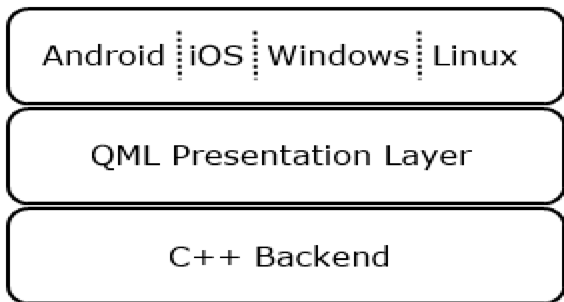


Figure 7: C++ backend handles the logic, with a QML presentation layer which is presented in a native style on the target platform.

4.4.2 C++ and QML

Early in the prototype stage C++ was used to develop an app using Qt Widgets. However the widgets are designed for static, desktop style user interfaces, while QML turned out to be a faster and more elegant way to implement a touch-based, animated mobile interface. However, the Qt libraries are implemented in C++ and QML can be integrated and extended with C++. The entry point for the app is also a C++ `main` function which sets up the event thread and loads the window with the initial QML component. The app also uses a separate worker thread written in C++ to generate tick events at a set interval - which uses the signals and slots mechanism explained in the next section on Qt Architecture - to trigger the event thread to make a server poll to periodically update the data.

C++ is a standardized general-purpose compiled programming language with a focus on performance and efficiency, and perhaps best known for being an object-oriented language. In the section on Qt Architecture we shall look briefly at the Qt style of C++, but for the interested reader there are numerous books, videos and tutorials teaching C++. It is also worth noting that although Qt started as a C++ GUI framework, bindings now exist for many other languages such as Ruby, Python, Java and Ada.

The newer International Organization for Standardization (ISO) C++11 standard from 2011 was used to take advantage of the C++11 Standard Library (std) `chrono` and `thread` class. These classes provide methods for creating a thread and putting that thread to sleep for a minimum duration specified in seconds using execution blocking. To enable C++11 gcc compiler support the following line was added to the Qt project (.pro) file.

```
CONFIG += c++11
```

The QML declarative markup language together with JavaScript functions was the primary language used to develop the Galaxy Portal app. This declarative language was discovered during the prototype stage of writing the app using Qt Widgets and C++, and turned on to be a very fast language to write prototype code in which could easily be re-factored into production code. Using the Qt Quick module to provide animations, transitions and more sophisticated effects such as shader and particle effects it was possible to rapidly develop an app with an entirely custom layout, but which would have the native style of the target platform.

At its most basic, QML consists of a series of elements that all inherit from the QML `Item` element and are defined by a set of properties. A QML `Item` is not visible, but has common properties for visual elements such as positioning and rotation, and QML Item elements can be grouped inside each other in a nested hierarchical tree structure. To give the reader a basic understanding a simplified version of the welcome screen is presented below.

```
Rectangle {
    width: Screen.width
    height: Screen.height
    Text {
        id: welcome
        horizontalAlignment: Text.AlignHCenter
        width: parent.width
        text: "Welcome"
    }
    Text {
        id: url
        anchors.top: welcome.bottom
        width: parent.width
        text: "Current Galaxy URL: " + dataSource
    }

}
```

`Rectangle` is an element that inherits from `Item` and provides an area to hold other items with a coloured background. To set the size of the rectangle the Qt Quick Window module provides `Screen.width` and `Screen.height` to set width and height of the rect-

angle to be the same as the current screen on which it is displayed. `Text` is another basic element that inherits from `Item` and in this example there are two `Text` elements contained inside the `Rectangle`. The first will be vertically centred inside its parent (the `Rectangle`), while the second text element has its top anchored to the bottom of the first. Both text elements are as wide as their parent and display a simple text string, which in the case of the second text element is concatenated with a QML property string. The way properties are bound in QML this means that when the `dataSource` property changes so will the text displayed in the `Text` element.

This small example only offers a glimpse of the QML language and is in no way intended to be a proper introduction or overview of even the most basic features of QML. It is, however, hoped that it will give the reader some understanding of how the language is structured to help follow the discussion in later chapters even if not everything is understood. Interested readers will find good documentation and many tutorials online, but as this is a relatively new language there is a lack of books on the subject.

4.4.3 Qt Architecture

The Qt framework consists of a number of modules of classes which can be added to a project to use that part of the framework. The basic module is Qt Core which contains the core Qt extensions to C++ such as meta-objects and the signals and slots mechanism used by objects to communicate. Other modules of interest is the Qt GUI module and the Qt Widgets module which adds C++ user interface elements. Instead of using the Qt GUI the Galaxy Portal app uses the Qt QML, Qt Quick and Qt Controls modules which provides classes for QML and JavaScript, and the supporting components and controls.

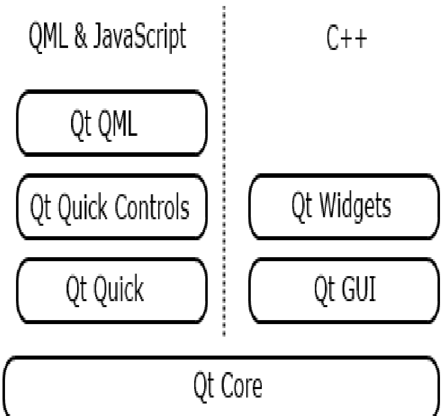


Figure 8: Shows the division between the Qt GUI C++ and Qt Quick QML modules, the latter of which is used in the Galaxy Portal app.

In Qt version 5.4 there are fifteen essential modules that are available on all platforms and at least twice as many modules providing a wide range of more specialised functionality that may only be useful for certain target platforms. None of these other modules were used in the app and will not be discussed.

All Qt classes are objects derived from the base class QObject. One of the distinguishing features of QObject is the signals and slots feature which allows communication between objects without using callback functions. This is achieved through meta-object code generated by the Qt meta object compiler (moc) which also enables other features such as run-time type information. The moc is run as part of the qmake build tool - itself an interesting cross-platform Makefile generator, but beyond the scope of this paper.

Since the QML Item type inherits the QtObject QML type, which in turn instantiates QObject, all QML elements are also derived from QObject. QML uses signals and slots for event notification, such as when the user clicks or taps a MouseArea element. It is interesting to note that signals can be used to communicate between QML and C++ code, as was the case for the C++ background thread we shall discuss in chapter six.

Qt is event driven which means that a main event loop is executed and runs for the lifetime of the application receiving events, such as user interaction with the UI. This event loop is initiated in the C++ `main` function contained in the main.cpp file. This file takes care of including the headers for the necessary Qt libraries such as QQmlEngine, but of most interest to us it creates the QGuiApplication object which contains the main event loop and QQuickView that displays the Qt Quick interface.

4.4.4 Qt Application Deployment

The Qt Creator IDE has many developer tools such as a code editor, debugger and integrated help system. It can load Qt projects with code, resources and build configurations grouped together and provides a wizard to assist with the setup of new projects and adding files to a project. The qmake utility tool is integrated with Qt Creator to build projects directly from the IDE or from the command line. A simple project text file specifies the Qt elements and files that need to be compiled when qmake is executed, and allows configuring the project.

Qt 5 handles multiple target platforms through the use of kits, where each kit can be configured for a particular target platform. Multiple target platforms can be configured and a project can be configured to compile any number of the target sets each time. In most cases the code must be built on the target platform, except for Android which can be built on any platform. Building an Android or iOS version requires an actual device

with debug mode enabled connected over a USB interface to the build machine, although it is possible to use virtual devices for testing. The Android Virtual Device manager is integrated into Qt Creator making it possible to create new virtual devices and easily set them or connected Android devices as targets for Qt Creator builds.

To deploy Android apps with Qt the location of the Android SDK and JDK must be specified in Qt Creator, and two additional components must also be downloaded and installed. The Android Native Development Kit (NDK) which handles using native code such as C++ in the Android app by compiling it into shared libraries (JNI) and the Java based Apache ANT build tool and a C++ compiler.

4.4.5 Considerations

There is extensive documentation and a large and active community supporting Qt development and Qt is widely used in software by many companies. One of the longest running such example being the open source cross-platform K Desktop Environment (KDE) which is a popular graphical desktop environment choice for Linux. There are also several books published (8) (9) (7) and many online video tutorials[53] which have been used to develop test apps and as a source for this section.

As has been discussed Qt is actively developed with new releases supporting new platforms and technologies and so is a good choice for an enduring cross-platform framework. The ability to develop applications under both open source and a commercial license could also be seen as a strength, but as we shall discuss further in chapter seven the open source GPL license creates legal conflicts with the distribution systems of some of these platforms.

During the initial investigation phase of the project several tutorial and test projects were built quite quickly using both Qt Widgets and Qt Quick, and deployed to Android. A basic login test was also developed using Qt Widget and is available on GitHub as Qt Galaxy Portal[54]. The final Galaxy Portal app released was developed using Qt Quick and will be discussed in detail in chapter six.

4.4.6 Conclusion Regarding Suitability for Galaxy Portal

The initial evaluation of the Qt framework based on the older Qt Widget C++ libraries was positive, but once the newer Qt Quick libraries and QML language was tested it became the favourite candidate for quickly writing user interface code tailor designed to modern mobile devices. With excellent support, an active community, multiple licensing

53 For example see https://www.youtube.com/playlist?list=PL2D1942A4688E9D63; accessed March 25 2015
54 See https://github.com/Tarostar/QtGalaxyPortal; accessed April 14 2014

options, good performance and regular new releases Qt was chosen as the best framework to achieve the most in a relatively short timespan with a fluid and good looking user interface.

Chapter 5

Galaxy

In the preceding chapters we have looked at mobile technologies and development and in the next chapter we shall look closer at how the Galaxy Portal app was implemented, but before doing that we shall in this chapter take a look at the Galaxy system which the app provides an interface for. In doing so we shall also explain the purpose for the Galaxy Portal app and what it sets out to do.

We first take a look at what Galaxy is and its web interface, and then we shall look at the Galaxy API which the Galaxy Portal app uses to communicate with the Galaxy servers. We shall also look at a previous attempt at developing a mobile interface for the Galaxy system and the Genomic Hyperbrowser as an example of systems that build on the Galaxy platform and what that means for the Galaxy Portal app.

5.1 Galaxy

Galaxy is an open, web-based platform for running analyses jobs in biology and medicine, and widely used in bioinformatics. The software makes it possible to run tools and workflows without programming experience and captures the information to make it possible to repeat and share analyses via the web. As discussed in the paper: "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences" Galaxy aims to provide accessible, reproducible and transparent genomic science (10).

Public Galaxy services running on high performance computing facilities are available[55] to anyone with Internet access and new Galaxy instances can be hosted on local servers

55 See https://usegalaxy.org; accessed March 15 2015

or using cloud hosting. Such local instances can run customised versions of Galaxy and allows custom development and new tools to be added. Galaxy is written in Python and the source code can be freely downloaded from Bitbucket[56] using the Mercurial version control system or as an archive.

The Galaxy Portal app aims to provide an interface for the Galaxy platform on mobile devices such as smartphones and tablets. Furthermore, it focuses on providing an easy means of monitoring the status of jobs, although it also makes it possible to take a look at the results of completed jobs and associated metadata. To understand why such a mobile specific interface is needed we must take a look at the existing Galaxy web interface and how the Galaxy platform is used.

The Galaxy web interface was designed when desktop and laptop computers were the primary method for getting online, but as we have seen mobile devices are increasingly being used and so we also have to consider the user experience on a small, touch-based screen with potentially limited bandwidth. While the Galaxy web interface is great for presenting a large amount of information in three columns on a wide screen, as seen in the image below, it is poorly suited for the narrow screen of a mobile device.

Figure 9: Galaxy web interface as shown in the Chrome browser on a desktop computer.

Selecting menus and other simple actions, such as logging in, is difficult, slow and error prone when attempted using a small touchscreen - which has a much lower level of precision than the mouse pointer which the Galaxy interface was designed for.

56 See https://bitbucket.org/galaxy/galaxy-dist; accessed March 15 2015

As Galaxy is typically used for data intensive analyses tasks that can take hours, or days, to complete a need was perceived for a more convenient way to check on the status of such jobs from a mobile device. This would allow researchers to monitor progress while away from the desk, and since the data from the completion of one process is often used to perform further analyses tasks, could also help improve efficiency.

5.2 Galaxy API

The Galaxy web interface is intended for human users, but the Galaxy service can also be accessed through the Galaxy Application Programming Interface (API). The Galaxy API is an attempt to transition Galaxy from the current architecture with a controller written in Python which serves up web pages, to a modern RESTful interface where data is served in JSON format. This was explained in a training day session during the 2013 Galaxy Community Conference held in Oslo[57]. This interface makes Galaxy data accessible by using a secret API Key unique to the individual user.

The API is represented as a set of controller modules that provides access to various aspects of the Galaxy system, such as the histories module which provides access to histories, or workflows module which provides access to the workflows. At the time of writing there were 36 such modules listed in the Galaxy API documentation[58], but the documentation is incomplete with information missing for many of the modules and it is entirely possible that there are additional API modules not referenced in the documentation.

The Galaxy Portal app uses three of the modules: authenticate, histories and histories_content.

The Galaxy API is designed according to the Representational State Transfer (REST) style to provide a RESTful API interface between client and server. The first thing to note is that the REST style sets out to separate the concerns of the client and server, which means that the Galaxy Portal app can be developed without worrying about how the data is handled on the server.

Looking at how the interface is organised into GET, PUT, DELETE and POST HTTP methods to static uniform resource identifiers (URIs) it uses a Resource-Oriented Architecture similar to what is described in chapter four of "RESTful Web Services" by Leonard Richardson and Sam Ruby (11). This is a somewhat more specific way to define the RESTful interface than how it was originally described by Roy Fielding in chapter five of

57 The talk on UI architecture and API starts at 22:45, https://vimeo.com/82179199; accessed March 20 2015
58 See https://galaxy-central.readthedocs.org/en/latest/lib/galaxy.webapps.galaxy.api.html; accessed February 28 2015

his dissertation “Architectural Styles and the Design of Network-based Software Architectures” (12).

A resource in this context is information, or a set of data, and every such resource has its own URI. As an example the histories module provides a URI to retrieve all (undeleted) histories: "`GET/api/histories`". To get a specific history the ID of that history is provided in the URI, for example: "`GET/api/histories/8b857f15403a37e6`". The examples provided are illustrations and do not show the actual URI. The full URI used is in fact the same as the uniform resource locator (URL) which also specifies the protocol used. Thus the URL is a type of URI which defines the protocol (in this case HTTP) to get the resource. Thus the full, or absolute, URI to get all histories from the usegalaxy.org domain for the `API Key 0f303101f8a957e35106c049f7ac38f9` would be: "`http://usegalaxy.org/api/histories?key=0f303101f8a957e35106c049f7ac38f9`". The final "key" parameter is a part of the URI and is provided to identify the user so that the service can return only the histories belonging to that user. This API key is automatically generated by the Galaxy service when the user logs in using the Galaxy Portal app and stored on the server and sent to the app. The app stores this key and uses it as part of the URI for all its communication with the Galaxy service. The user can at any time change this API key by generating a new key using the Galaxy web interface and then either typing or pasting the new key into the API key field in the app, or simply logging in again. This highlights that multiple URIs can point to the same resource, although the old URI will of course no longer work since that API key was deleted.

Although the URI identifies the resource it does not specify the format and language of the resource. It would of course be possible to encode this information into the URI, but the Galaxy API uses content negotiation by specifying the desired data representation in the request header. In the case of Galaxy Portal the Galaxy service requires us to specify the language and content-type. While English language data formatted in JSON (the JSON format is talked about in chapter six) is currently the only available representation, the Galaxy API still requires these to be specified and thus makes it possible in the future to provide additional representations in other languages and new formats. See appendix B.1 for the code where these request headers are specified in Galaxy Portal.

Another important aspect of a RESTful API interface is that it is stateless. What this means is that every request happens in isolation with all the information needed in the URI. Thus neither the server nor the client needs to worry about preceding requests and can simply send a URI request without worrying about state. This is part of what makes RESTful interfaces so easy to scale through methods such as load-balancing and caches

which would be a lot harder to do if taking into account state. More relevant to app development is that eliminating state from the protocol also makes it much simpler and eliminates many types of errors that could occur from not tracking state correctly and getting "out-of-sync" with the server. Just because the RESTful API interaction is stateless does not mean that the client or server do not have a state. The app client obviously has a state as the user navigates from a history, to the jobs in a history and potentially opens up one of those jobs to retrieve the dataset for that job, but the resource request is entirely defined by the URI the client provides to the server and not by its state or previous such requests.

As mentioned at the start, the Galaxy API defines four methods for interacting with resources: GET, PUT, DELETE and POST. These methods are defined by the HTTP protocol and since to be RESTful the API must provide a uniform interface anyone familiar with these methods will already have a good idea of how they work. Indeed we can see from the API documentation that GET is used for read operations to retrieve data, DELETE to delete resources and PUT to create or update a resource. POST is used for various other things such as appending to the database by creating new histories and providing data such as for scheduling a workflow to run.

When a request returns from the server it contains a JSON formatted response, but also a HTTP status code which follows the Hypertext Transfer Protocol (HTTP/1.1) RFC 2616 standard[59]. Briefly this means that a successful request has a status of 2xx, typically 200, and 4xx or 5xx for client or server errors respectively. This allows the Galaxy Portal app to distinguish a successful request from a failed request and take the appropriate action such as notify the user with an error message.

This primer should be sufficient to understand how the Galaxy Portal app communicates with the Galaxy service, but those who wish to know more can look up the "RESTful Web Services" book mentioned earlier or explore the wealth of information online.

5.2 Previous Solutions, Wrappers and Extensions

In this section we shall take a look at Galaxy mobile, which was an early attempt to provide a mobile interface that did not use the Galaxy API. We will also take a look at Galaxy BioBlend which wraps the Galaxy API and why we did not use it. Finally, we shall look at the Genomic Hyperbrowser as an example of an extension to the Galaxy framework and what that means for the Galaxy Portal app.

59 See http://tools.ietf.org/html/rfc2616; accessed Feb 28 2015

5.2.1 Galaxy Mobile

While researching this thesis it was discovered that there had previously existed a Galaxy mobile interface. This was added by the Galaxy team in 2009 to target the iOS platform which was then relatively new on the market. The Galaxy API was not used, but instead a custom controller written in Python was used to access the Galaxy system. The front-end was implemented in HTML, CSS and using the jQTouch JavaScript library.

The mobile interface was recently removed because it had not been maintained and was not being used. It only supported a few basic features such as logging in and retrieving a list of history items and the lack of maintenance means it had a number of dead links. It may be that being implemented so long ago it was ahead of its time and the actual need for such a mobile interface. The original author, one of the core members of the Galaxy team, was contacted for further comments, but although he did indicate that he would respond, any answer did not arrive in time to be included in this text.

5.2.2 Galaxy BioBlend

Apart from the Galaxy framework itself, perhaps the most prominent software that takes advantage of the Galaxy API is BioBlend (13). BioBlend was developed to provide a high-level interface for bionformaticians to automate data analysis that would be inefficient to do using the Galaxy web interface. It is a Python library that wraps the Galaxy API, as well as the Galaxy cloud infrastructure (CloudMan) API, and provides a high-level Python interface which handles the HTTP requests, data serialization and deserialization from the JSON format, and errors.

With the new BioBlend.objects component (14) BioBlend also provides an object-oriented interface which isolates the client from changes to the Galaxy API by returning defined objects, but would still require BioBlend to be upgraded.

BioBlend was considered as a potential alternative to using the Galaxy API directly - especially when looking at the Python framework Kivy. Released under the MIT open source license, which grants unrestricted use of the software, there was no license issues in doing so. However, having to install BioBlend and the Python libraries it depends on would have added additional complexity to deploying a cross platform app and it was noted that while BioBlend has been tested on Linux, Macintosh and Windows there might be issues with using it on iOS, Android and other mobile platforms potentially complicating and limiting the cross platform potential of the Galaxy Portal app. Most importantly there was little perceived benefit from using the BioBlend wrapper as talking directly to

the Galaxy API should be more efficient and JavaScript is very capable when it comes to making HTTP requests and parsing JSON formatted data.

5.2.3 Genomic Hyperbrowser

"The Genomic HyperBrowser web server provides a broad suite of functionality for rigorous statistical analysis of genomic data" (15). The Genomic Hyperbrowser is mentioned here as an interesting platform which builds on the Galaxy platform to provide additional functionality. As stated in the same paper quoted above: "The tools share an underlying analysis code base, which is open-source and tightly integrated with the Galaxy framework for handling of web access, users and data. Through the integration with Galaxy, the standard Galaxy tools are also available and can be used together with the HyperBrowser-specific functionality." The interesting aspect here is that since the Hyperbrowser builds upon the Galaxy framework the same API interface is also available. This means that the Galaxy Portal app can also be used to browse histories and monitor jobs on the Genomic Hyperbrowser, and potentially any other future platforms that build upon the Galaxy framework.

Unfortunately, testing revealed that the Genomic Hyperbrowser was running on an older version of the Galaxy source with a limited API support. It was still possible to retrieve histories by manually typing or pasting the API Key from the Genomic HyperBrowser into the app, but the login functionality and showing the full datasets for jobs did not work. At the time this text was being written a new version of the HyperBrowser was being developed built on the current version of the Galaxy framework. Early user access was obtained and the API tested confirming that the API interface now is fully supported, although the login functionality could still not be tested as it was only available through an additional web login with a redirect making proper testing with the app impossible.

Part II

Implementation

Chapter 6

Implementing Galaxy Portal

In this chapter we look at the Galaxy Portal app and how it was implemented. Some code examples are provided in this chapter to illustrate concepts and methods. These code snippets are all taken from the Galaxy Portal project, sometimes with minor modifications for clarity, to help the reader become familiar with the actual code used. The code is not explained in detail and any comments have been removed to keep the examples concise and readable. See appendix B for the complete code and binary releases.

As can be seen in figure 1 of chapter one, the Galaxy Portal app consists of an action bar at the top with a main view below that displays data and settings. The action bar helps the user orient by displaying the title of the current view along with an arrow to go back to a previous view, number of items in any displayed lists and icons for any other available actions in the current view or selected field. This results in a consistent interface where the user drills down into greater detail by selecting items in the view and then returns back up using the back arrow. For a more in-depth description and illustration of the app functionality consult the user guide referenced in appendix A.

6.1 Code Development Practices

Coding style and development practices often come down to personal preferences, or the preferences of the team, previous code authors or as advised by a relevant style guides. The choices made for Galaxy Portal were based on personal development experience and the coding styles promoted by the Qt libraries and associated framework.

6.1.1 Style

Style covers all aspects of how the code is written and formatted for human eyes – as opposed to the valid syntax required to be understood by the code compiler. The purpose of maintaining a consistent style is to improve readability and so make writing code easier, more enjoyable and less error prone.

The code written uses the capitalisation style referred to as camel case where the first letter of each compound word describing a method or variable starts with a capital letter. Following the style chosen by the Qt framework classes, components and elements all start with a capital letter, whereas methods, variables, properties, types and ids all start with a lowercase letter while using camel case for any compound words.

Names of methods, variables and other code elements are always chosen to be descriptive, even when that means using longer and less elegant descriptors. The idea is to write code that is self explanatory and comments are used only sparingly where additional explanation is necessary or it improves readability and ease of comprehension. The advantage of using this approach is that as code changes so does the description, whereas when relying on comments and documentation these may not be updated when the code changes. Having comments and documents that mislead is even worse than having no explanation at all.

Another important style imposed on the code by the Qt Creator IDE is the colour coding where variable names, methods, strings and other types of code are rendered in a particular colour. This is a common feature in most code editors exactly because it is so useful and it makes the code far easier to read and many types of errors become immediately obvious, such as the entire text turning green because of a speech quote missing to close off a string.

6.1.2 Methodology

As the coding was done while learning the languages, tools and framework it felt natural to use some of the agile software development methods such as using short iterations to release working software early and often, and to refactor the code to incorporate new features and better solutions discovered in the process. This approach was very useful for learning quickly what worked and what did not.

The intention was also to get the app into the hands of actual users for testing relatively early in the process. This worked well on the Android platform. However, it proved difficult on the iOS platform where the only way to get the app to potential iPhone and

iPad users was to make them members of the team in iTunes Connect[60] as testing was restricted to a maximum of 25 internal testers. The result was practically no feedback on the iOS platform as the app was only re-licensed and approved by the Apple review when already on release version 1.

The modular structure of Qt and the declarative QML language lent itself to fast iterative cycles with frequent refactoring. To keep the code manageable as it grew, separate functionality was kept in separate files. It was also generally avoided to write the same code in several places and instead such code was collected in QML components and JavaScript functions to be used throughout the app.

The focus was to get things working in a simple way that would lend itself to being developed further rather than trying to optimise performance.

6.1.3 Source Control

Git was used for source control and all source is stored in the openly accessible GitHub source repository. This allowed the same source to be downloaded to different machines using different operating systems. GitHub did not only function as a source backup, but also as host for the binary releases for each version along with the source needed to compile that version and thus fulfilling one of the key requirements of the GPL license.

Source control was an essential part of the development process as well as it provided easy access to previous versions and even served to help document the code as each code commit is done along with a description of the changes.

6.1.4 Working with large third party frameworks

Anyone who has worked with software development for any length of time knows that flaws and mistakes are inevitable, and so it is essential to write code that will degrade as gracefully as possible and report the cause as accurately as possible. However, when developing using a third party framework like Qt is also becomes important to be aware of bugs, limitations and known issues with the version used. The Qt source is massive and in constant development, always adding new features and support for new platforms and technologies, and thus prone to bugs and unexpected side effects. During the development of Galaxy Portal several such bugs were discovered. The benefit of the open-source nature of the framework and large community surrounding it is that most bugs are discovered and reported. Qt has a bug report system to enable tracking and fixing such bugs, but this also provides information to help implement workarounds.

60 itunes Connect is the web console for developers to manage apps for the App Store, as well as other Apple marketplace content.

For example, reported bug QTBUG-21706[61] meant that the response text was being cleared for an XMLHttpRequest when receiving an error before reaching the final "DONE" state. Thus to be able to display error messages related to the polling the error message had to be set during the loading phase. Another known bug in the Qt Quick Controls module, QTBUG-36849[62], reports a false error: "QML Text: Binding loop detected...". No workaround was needed for this bug, but being aware of its existence prevented time being wasted on trying to figure out the error.

6.2 The QML Structure

As we saw in our brief introduction to the QML language in chapter four we can build a user interface by placing QML elements, such as rectangles and text, and grouping them together in a tree structure. This becomes even more useful when we realise that new components can be created by putting a composition of elements together and simply saving them in a file with the .qml extension or giving the root element an id. Thus we can, for example, build a custom button composed of rectangles, text, images, animations, notification handlers and even JavaScript functions and then use that button throughout our code simply by referencing the name of the file, e.g. `MyButton` if the file was *MyButton.qml*.

This sort of functionality is not unique to QML, but its usefulness comes from how easy and intuitive it is to build up the user interface. We mentioned in chapter four that QML is a declarative language, and this focus on structure without worrying too much about the application logic is a big part of what makes it such a fast prototyping language.

6.3 The RESTful API interface

We looked at the Galaxy RESTful API interface in chapter five and as we have seen we need to make HTTP requests to various URI to retrieve data and perform actions such as logging into the Galaxy system. To do this the `XMLHttpRequest` object is used, which is a World Wide Web Consortium (W3C) standard. As this is a common and widely used method we shall not elaborate on it here, other than to mention that it makes an asynchronous call so that when data is received the event handler assigned to `onreadystatechange` is called.

The Qt implementation of `XMLHttpRequest` does not implement the cross-domain

61 See https://bugreports.qt.io/browse/QTBUG-21706; accessed March 30 2015
62 See https://bugreports.qt.io/browse/QTBUG-36849; accessed March 30 2015

blocking defined in the standard which is part of pretty much all browsers to prevent exploits of security vulnerabilities in JavaScript. This is a crucial point as cross-domain blocking would prevent a call from the app to any Galaxy instance as they would not be on the same domain. Since not implementing this feature is intentional and documented the risk is low that such blocking would be added in the future.

6.3.1 Histories and Jobs

Galaxy organises jobs into histories, with each history being a collection of content - typically datasets for jobs - and for simplicity we refer to them simply as jobs. Thus once a user is logged in using "GET/api/authenticate/baseauth" to retrieve the Galaxy API Key we can use that key to get a list of all histories with "GET/api/histories". This retrieves sufficient data to display the initial list of the user's histories and critically provides the unique id of each history. This unique id allows us to construct the unique URIs for a specific history and retrieve the list of jobs for that history. This list of jobs contains the unique id for each job to retrieve this dataset, but the list also contains a copy of the job's metadata such as name and status. This is very useful as it means that we can display the list of jobs for a history with the current status of each job without having to making additional requests for the data of each job.

To save bandwidth and improves performance an additional request to retrieve the actual dataset for the job is only made when the user wants to take a closer look at a specific job - by tapping or clicking on it in the list. Just like histories are containers for jobs, each job item is a container for the dataset for that job - including metadata about the job.

6.3.2 Job Status

Reporting the job status is the focus of the Galaxy Portal app and so is prominently shown with colour coding and by showing a status field for any item selected by flipping the item around to show additional data. The twelve possible status states as listed by the "state_details" field in the Galaxy API are shown in the figure 10 with the corresponding colour coding in Galaxy Portal.

Job Status Colour Coding

Status	Colour
Ok	ivory
Running	aquamarine
Paused	yellow
Queued	sky blue
Error	red
Discarded	gray
Empty	lemon
New	snow white
Resubmitted	cyan

Figure 10: Job status colour coding with the status on the left and colour on the right. Taken from my Galaxy Portal user guide.

"Ok" is the status for any completed job that did not have errors and uses the default ivory colour. We are especially interested in "running" jobs which are colour coded in aquamarine green (shown below), and errors which are indicated in red.

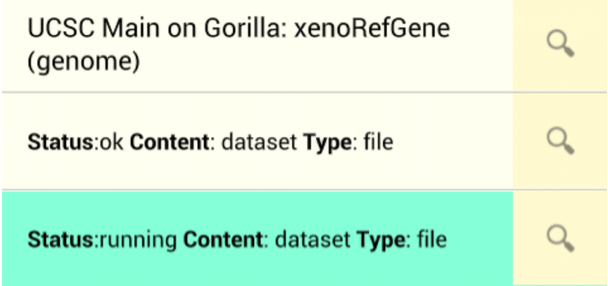


Figure 11: Part of a job list from Galaxy Portal showing the status for two items that have been "flipped" around by tapping them, one which is a running item in aquamarine.

To draw attention to status updates the app plays a notification sound whenever the status changes and a distinct sound when a job that was running completes or runs into errors.

6.3.3 API Interface Interaction

All communication with the Galaxy API is handled through a single JavaScript function. This JavaScript function uses the Qt QML `XMLHttpRequest` to retrieve data as shown in appendix B.1. XMLHttpRequest is a widely used JavaScript object, but the Qt QML version does not implement any timeout support and JavaScript methods such as setTimeout are not available. To overcome this a QML `Timer` element is created which triggers after a set

interval, aborts and reports the timeout by calling the `onReady` JavaScript function without any parameters. The code below illustrates how this QML object is dynamically created inside the JavaScript function from a text description using `Qt.createQmlObject`, and how a signal - in this case “triggered” - can be connected to a JavaScript function.

```
var pollTimer = Qt.createQmlObject("import QtQuick 2.3;
                        Timer {interval: 5000; repeat: false;
                        running: true;}", parentID, "PollTimer");

pollTimer.triggered.connect(function() {request.abort(); onReady();});
```

The JavaScript function can be called from any QML component by importing the JavaScript file (utils.js) and then calling the function as shown here.

```
Utils.poll(source, onReady, details);
```

An optional fourth parameter lets me set the request header as an authorization header so I can also use the function in the `GalaxyKeyBaseAuth` component to log in and retrieve the API key. When set the standard JSON `Content-type` header specifying “application/json” is replaced with a base64 encoded header containing username and password. The resulting function call is shown here.

```
Utils.poll(galaxyUrl.text + "/api/authenticate/baseauth",
      onReady, galaxyAuthentication, "Basic " +
      Qt.btoa(baseAuthUsername.text + ":" + baseAuthPassword.text));
```

6.3.4 JavaScript Object Notation (JSON)

The data received by the `XMLHttpRequest` from the Galaxy API uses the JavaScript Object Notation (JSON) format[63]. This is a widely used format with a simple structure similar to a JavaScript object. It holds history items in a list structured like an array, with the data stored in key-value pairs as shown on the next page for a list with only a single history.

63 See http://www.json.org; accessed February 1 2015

```
[
    {
        "deleted": false,
        "id": "f597429621d6eb2b",
        "model_class": "History",
        "name": "MyHistory",
        "published": false,
        "tags": [
            "Test"
        ],
        "url": "/api/histories/f597429621d6eb2b"
    }
]
```

This means that I can use the standard JSON parser to turn this JSON formatted data into a JavaScript object.

```
var objectArray = JSON.parse(json);
```

The “json” value parsed in the above example is a QML `property` of the `string` type. This property is set in the `onReady` function called by the asynchronous event handler for `onreadystatechange` set in the JavaScript poll function above. All QML properties have a signal emitted when changed, so when the “json” property is updated with new JSON data it emits a signal. The corresponding slot that receives this signal is `onJsonChanged`. This is a convention where the slot for any property is always “on<property name>Changed”. Since this slot is called every time there is new data in the “json” property the model[64] update is done from this slot.

The model update in the JSONListModel compares the existing model with the received data and removes, appends, inserts or moves items to match the new data. This means that new data can be incorporated into the model without reloading the whole list and so not interrupting the user interacting with the list. The keys in the JSON data I want are known from the API document, such as the “name” key for item title, so extracting the data from the JSON and inserting it into the model is done by simply referring to the key corresponding to the model field. See appendix B.2.

The `DetailView` component also uses a model which is updated when its “json” prop-

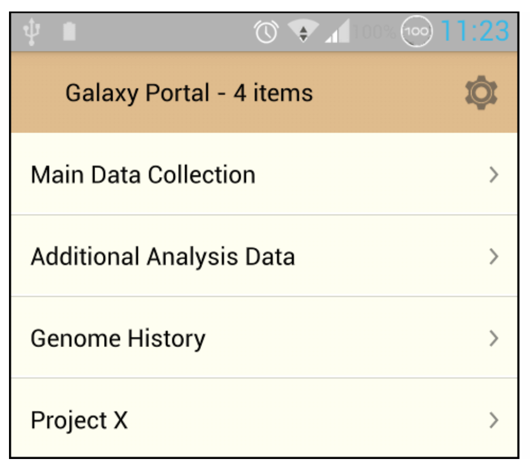
64 See section 6.4 for an explanation of how models are used in Qt.

erty value changes, but since I want to list all available data for the dataset I do not want to hardcode the keys. By instead iterating through every key in the object I ensure all available values are displayed without needing any code changes or configuration changes for datasets with different key values. To do this I check if the value is not empty and has a type of string, boolean, number or symbol and if so insert it into the list model with the key as its name. This process ensures I only attempt to show values that will display correctly and is shown below.

```
var jsonObject = JSON.parse(jsonData)
for (var key in jsonObject) {
        if (typeof jsonObject[key] === "boolean" ||
                typeof jsonObject[key] === "number" ||
                typeof jsonObject[key] === "string" ||
                typeof jsonObject[key] === "symbol") {
                if (jsonObject[key].toString().length > 0) {
                        detailListModel.append({
                        "fieldName": key.toString() ,
                        "fieldData": jsonObject[key].toString()});
                }
        }
}
```

6.4 Model-View Design Patterns

The Galaxy Portal app is organised to a large extent by lists of items. On the settings page new Galaxy connections can be saved to a list from which the user can select a Galaxy instance to connect to without having to supply username and password again. Once logged in to a Galaxy instance the user will be presented with a list of history items. From this list the user will be able to select a history and get a list of jobs for that history. The job can be flipped by clicking on it in the list to show the dataset for that job and this set can be configured by selecting from a list of the available fields on the settings page. Clicking the magnifying glass on a job brings up a list of its entire dataset. The screen-shots from the Galaxy Portal app below shows a history list on the left and list of jobs on the right.



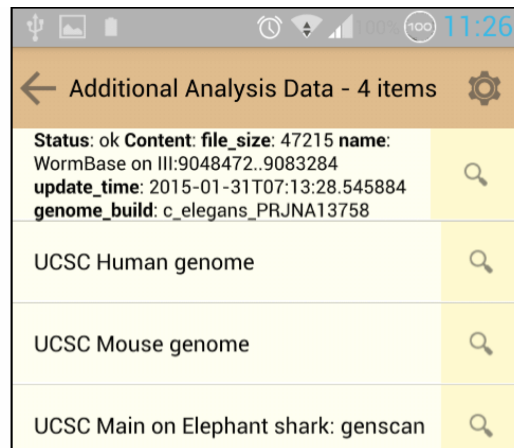


Figure 12: List of Galaxy histories on left, and list of job items on the right with the top item flipped to show additional data (data shown is configurable).

6.4.1 The Qt Model-View-Delegate Pattern

To manage the data for these lists separately from how it gets displayed the Qt Model-View pattern is used. In chapter seventeen of "An Introduction to Design Patterns in C++ with Qt 4" the Model-View pattern is described as: "techniques of separating the underlying data (the model) from the class that presents the user with a GUI (the view)" (9).

By separating the data model and how it is presented to the user we get code that is less complex and easier to maintain. This design pattern is sometimes referred to as model-view-delegates as instead of using the classical model-view-controller (MVC) pattern, where the controller is responsible for communicating changes to the data in the view and model, Qt uses a delegate which encapsulates the data from the model and presents it to the view as it should be shown.

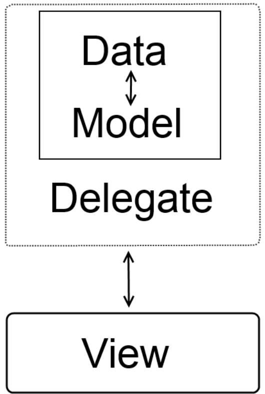


Figure 13: Illustration showing how the delegate encapsulates the model with the data and presents it to the view for rendering.

6.4.2 Decoupling Model and View

The purpose of model-view patterns is to decouple models and views to make it easier to make changes to either without affecting the other, something which is also very convenient when creating mock models during unit testing. There are many variations on how this separation of model and view is achieved and many frameworks use these patterns somewhat differently. This can pose a challenge when two frameworks which both label their architecture design as using the MVC pattern may not behave in the same way. Clearly the use of the term delegate makes it clear that Qt uses its own model-view pattern, just like Microsoft architects describe their brand of the model-view pattern as model-view-viewmodel (MVVM).

The model-view-presenter (MVP) pattern is another classical pattern which attempts to remove logic from the view, and does not allow the view to update directly from the model like the MVC pattern typically does. All communication between the model and the view must go through the presentation layer. Qt's model-view-delegate architecture has some similarities to this in that the delegate handles all the communication between view and model. However, the delegate is also responsible for rendering the view by positioning and formatting elements while the view is in many cases reduced to little more than a container. Thus unlike in the MVP (or MVC) pattern the delegate handles both the input logic usually handled by the presentation layer or controller, and the presentation of the data typically done by the view. This tight coupling of event logic with view logic makes it easier to quickly create a functional user interface, but ties the implementation to using the built-in event handling logic.

6.4.3 Implementing the Model-View-Delegate Pattern

QML provides several models for holding data, such as a list or tree structure, and additional models can be created in C++. In Galaxy Portal I used the `ListModel` type and took advantage of the fact that items can be inserted using JavaScript, which then also defines the fields of the model, as shown below for the `DetailView` component responsible for displaying all the job's dataset fields.

```
detailListModel.append({"fieldName": key.toString(),
                        "fieldData": jsonObject[key].toString()});
```

The two main list views - the history and job list - both use the custom `JSONListModel` component which encapsulates the `ListModel` to provide functionality to update the

model. Since the Galaxy API data is formatted as JSON, JavaScript is used to parse the JSON object into an array and insert each item into the `ListModel`.

When the model is updated the delegate determines the text elements in the list view and which fields from the model structure are shown and how they are formatted - including any areas that can be tapped or clicked and other user interaction.

The lists in Galaxy Portal are all presented using the QML `ListView` element, but they could also have been shown as grids using `GridView`. The `ListView` defines the model and delegate used, and sets the position, size, scrolling and other properties that apply to the list view as a whole.

For the list of histories the delegate refers to `HistoryDelegate`, which is a custom QML component implemented in its own QML file with the layout for each list item as shown in the list of Galaxy histories on the left in figure 12. Below is the code for the `ListView` as discussed here.

```
ListView {
        id: historyListView
        width: main.width
        height: main.height - mainActionbar.height
        model: jsonHistoriesModel.model
        delegate: HistoryDelegate {}
        clip: true
        boundsBehavior: Flickable.StopAtBounds
}
```

This is the main `ListView` of available histories presented when connecting to a Galaxy instance. The `ListView` for displaying jobs is almost identical, but uses its own model and the `JobDelegate` component as its delegate. It is interesting to see that the entire code for the `ListView` is quite short. Of course the real complexity is in the model and delegate, each implemented in its own QML file.

Without going into too much detail the `HistoryDelegate` component contains QML elements, such as `Text` to display the item title, which refer directly to the model as defined in the `ListView` to retrieve the data to display. Thus we can see that the view is just a simple list container which defines a model and delegate, where the delegate communicates directly with that model to retrieve data and is responsible for how it gets displayed in the view.

6.5 Navigating Views

The app is divided into different views that display various lists of data, such as the history list, job list and dataset list for a job, and there are other views that shows the settings, a list of saved Galaxy instances and also a view dedicated to showing a single data field for a job to easier look at tables and other fields with large amounts of data.

To present the correct view based on the current app state and user action, and to transition between these views in a smooth and intuitive way a number of different methods are used.

6.5.1 Show and Hide Components

When the Galaxy Portal app is started it will initially show an action bar at the top with a welcome screen below as shown in figure 14.

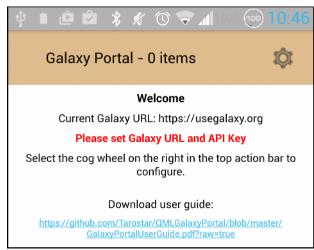


Figure 14: Welcome screen when starting Galaxy Portal for the first time.

This welcome screen provides some simple directions and shows the URL for the current Galaxy instance as well as a status message such as missing API key or connection errors. This welcome screen is a simple QML component named `Welcome` consisting of a `Rectangle` element with the `Text` elements as children. It is only shown when we are on the main page (no special state is set) and there are no history items in the model.

```
visible: (!challengeDialog.visible &&
        jsonHistoriesModel.count === 0 && main.state === "")
```

We check if the `challengeDialog` is visible as a condition for showing the welcome view. This is the identifier for a `PasscodeChallenge` component which when enabled in the settings presents a view with a passcode challenge when starting Galaxy Portal.

```
visible: passcodeEnabled
```

If the user fails to type the correct passcode an option to bypass it is offered, but doing so deletes all user data such as saved instances, username and API keys. This component is the first element drawn in the `main` QML file loaded at start-up and set to fill the parent element (done using: "`anchors.fill: parent`") so that when it is visible it is the only thing shown.

Both the welcome view and the passcode challenge view then are simply shown or hidden by using the "visible" property of the rectangle element, which then also hides or shows all child elements. This makes sense for views that are shown only under special circumstances and to the exclusion of everything else, but for the other views we shall look at more sophisticated methods of controlling the shown view using both states and the QML `Loader` type.

6.5.2 Animated State Transitions

Let us first look at how a state can be used to transition between views using an animation. To better understand how we transition between views let us first look at how the components are organised. In the `main` QML file a `Column` QML type container is used to draw the QML components vertically beneath each other. The first component is the `ActionBar` which holds the view title and action buttons along the top, followed by the `Welcome` component that we have looked at, and then a `Row` QML type container which is only visible when the `Welcome` component is not shown.

The `Row` container organises its children horizontally, and contains the two `ListView` elements for the history list and the job list associated with the selected history. The width of each `ListView` is set to be equal to the width of the app, so that initially only the history list is visible in the default state, while the job list is hidden off the screen. When a history item is drawn an arrow is shown on the far right (see figure 12) and the delegate for each history item has a `MouseArea` element which receives taps or mouse clicks and sets the current selected history and then triggers a state change by setting the state to "historyItems". A QML state can have its own configured properties, but in this case all we do is set the "x" property of the `Rows` QML type to be the width of the screen and thus transitioning the view along the x-axis as shown in figure 15.

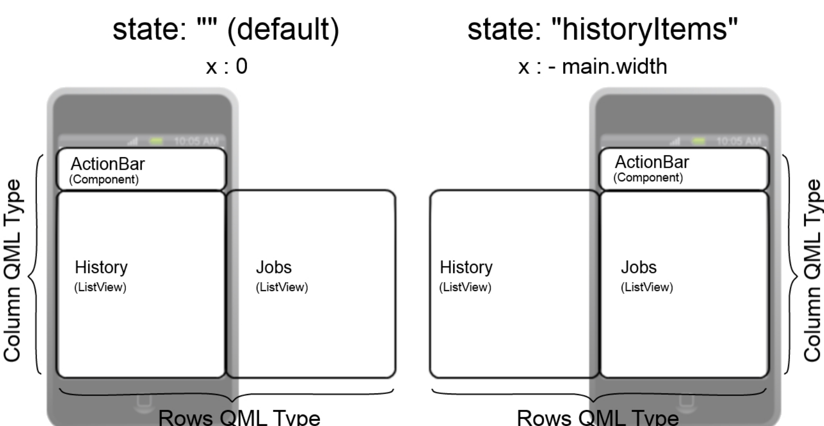


Figure 15: Illustration of children of `Column` arranged vertically, with children of `Rows` arranged horizontally. Default state on the left. On the right the x-axis offset is set to the width of the screen for the "historyItems" state to transition the jobs `ListView` on to the screen. The `Welcome` component would fill the screen below the ActionBar component when visible and so push the Rows container with the History and Jobs off the screen, but is not normally visible and so not shown for simplicity.

The code for the states is defined as shown below, and note that the target referred to is the id of the `Row` QML type meaning that the state changes are applied to the `Row` element.

```
states:
        State {
                name: "historyItems"
                PropertyChanges {
                target: screenlayout
                x: -main.width
        }
}
```

The actual transition is handled with a QML `Transition` using a `NumberAnimation` QML type which has the x property of the `Rows` QML type as its target and a duration of 1000 milliseconds which means it will take one second to go from the start value to the new target value for x as set by the state. An `OutCubic` easing curve type is specified, which means that the animation will start out fast and then gradually slow down as it

gets closer to the target providing a smooth transition.

```
transitions: Transition {
        NumberAnimation {
                target: screenlayout
                easing.type: Easing.OutCubic
                property: "x"
                duration: 1000.0
        }
}
```

To get back from the jobs list the user simply taps the back arrow in the action bar, which sets the state back to an empty string indicating the default state, and the view transitions back to the main history list using the same animation in reverse. This provides a good user experience by animating moving smoothly forth and back between lists.

6.5.3 QML Loader

A QML `Loader` type is used for the remaining views to dynamically load components on top by setting the z-axis property for the component item to one when loaded by the `Loader`. This powerful feature is very simple as shown by this single line of code that implements the loader.

```
Loader {  z: 1; id: mainLoader }
```

To load any component as the current view all that is then required is to set the source property. For example, the settings view is loaded when tapping the cog wheel in the action bar with the following line of code:

```
mainLoader.source = "Settings.qml"
```

Then when tapping the back arrow in the action bar the source is set to the `backSource` property defined in the action bar, which defaults to a blank string and thus simply unloads the current component returning the user to the default view - which is either the history list or job list depending on the state.

It is also possible to call JavaScript functions using the QML `Loader` and this is used when the `InstanceList` component displaying the list of Galaxy instances is loaded to call a function to populate the model with all the Galaxy instances that have been saved by the user.

6.6 Device Screen Scalability

When developing for a range of devices with different screen sizes and display resolutions it is important to consider how the user interface will scale. User interface elements such as list items and buttons cannot be too small since the primary method of interaction is touch-based, but at the same time the space must be used efficiently to present as much information as possible on a small screen.

Measurements can simply be done in pixels so that when the display resolution increases and more pixels fit on the screen there is more space to display list items, text and other user interface elements, but this would mean that everything would be smaller on a device with a higher resolution unless the screen also gets bigger. Since a mobile app is expected to look the same on a wide range of screen sizes and resolutions this will not work well. Instead, we need a device independent pixel count and to do that we need to know the number of pixels in proportion to the screen size.

6.6.1 Pixel Density

Since screens are typically measured in inches this is usually calculated as dots per inch (DPI)[65]. Since the screen size is measured diagonally from corner to corner we get a right angled triangle, and so can use the Pythagorean Theorem to find the corresponding pixels across the diagonal. We can then divide this number of pixels by the screen size and get DPI.

$$\frac{\sqrt{\text{pixel-width}^2 + \text{pixel-height}^2}}{inches} = DPI$$

On an Android device independent pixels (DP) are used which are relative to a 160 DPI screen, while on iOS points (PT) are used where one point is 1/72 of an inch. However, instead of relying on a platform dependant measurement Qt has a `Screen` QML type which has a pixelDensity property defined as the number of physical pixels per millimetre for the screen the element is currently being displayed on. To get access to this the

65 Dots Per Inch (DPI) originated from print, and it is arguably more correct to use the term Pixels Per Inch (PPI), but as nearly all documentation refers to DPI we shall do so here as well to avoid confusion.

QtQuick.Window module must be imported with the line: `import QtQuick.Window 2.2`.

The recommended minimum sizes for touch-based UI elements varies from 7-10 mm in the Android design style guides[66] and 9 mm minimum in the Microsoft Phone guidelines[67], to 15.5 mm (44 points) in the iOS human interface guidelines[68]. To satisfy these requirements, as well as user preferences, the exact scale of the interface can be adjusted in the settings up to three times the standard size in the following multiples: x1, x1.5, x2, x2.5 and x3. Based on smartphone testing the standard size is set to 10 mm with most margins being 3 mm, but to make it easy to change these in the future the values are set as properties in the main.qml file and referenced throughout the project as shown here.

```
property int mmItemHeight: Screen.pixelDensity * 10 * scale;
property int mmItemMargin: Screen.pixelDensity * 3 * scale;
```

6.6.2 Resolution Categories

All icons are also available in different resolutions to ensure a high fidelity image appropriate to the device resolution. As adding multiple graphics for every icon is a lot of extra work and increases complexity and platform dependence it might be worth exploring alternative approaches, such as using svg vector graphics, but for the current version of Galaxy Portal the four size categories recommended for Android were used. The `Screen.pixelDensity` measurement is used to select the appropriate icon resolution and the categories are as shown in the table.

Resolution Category	Screen Pixel Density	Icon Pixel Size
medium (mdpi)	< 6.3	32 x 32
high (hdpi)	< 9.3	48 x 48
extra-high (xhdpi)	< 12.6	64 x 64
extra-extra-high (xxhdpi)	>= 12.6	96 x 96

Table 2: The four categories of icon sizes selected according to the pixel density of the screen, where pixel density is pixels per millimetre.

66 See http://developer.android.com/design/style/metrics-grids.html; accessed February 8 2015
67 See https://msdn.microsoft.com/en-us/library/windows/apps/hh202889%28v=vs.105%29.aspx; accessed February 15 2015
68 See https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/LayoutandAppearance.html; accessed April 8 2014

There is one directory for each resolution category with the appropriate icon or image. By using the pixel density of the device the correct directory can then be selected to load all graphics from. To manage the resource in a device independent manner they are all stored in a resource file, and so the path to, for example, the medium resolution category is: "qrc:/resources/resources/icons/mdpi/".

When the app loads the path to use is set once in the `iconRoot` read-only property in "main.qml" by picking out the right directory based on an index value returned by the simple JavaScript function `getResolutionIndex` - which simply returns the appropriate index based on the pixel density. The QML properties are defined as shown here.

```
readonly property var res: ["mdpi","hdpi","xhdpi", "xxhdpi"]
readonly property int resIndex:
            Utils.getResolutionIndex(Screen.pixelDensity)
readonly property string iconRoot:
      "qrc:/resources/resources/icons/" + res[resIndex] + "/"
```

This `iconRoot` property is used everywhere in the project to draw the icon appropriate for the current device resolution. Furthermore, when a button is pressed a different lighter version of the same icon is drawn to indicate that the press was registered and so provide a good user experience. This is done with a second set of icons of the same title with the string "_pressed" appended to the end as shown for the image source of the zoom button image below.

```
source: mouseArea.pressed ?
                  iconRoot + "ic_action_zoom_pressed.png" :
                  iconRoot + "ic_action_zoom.png"
```

6.6.3 Device Independent Font Size

For QML Text elements the Qt pointSize property was used for fonts to set them independently of display density. This helps fonts look the same size regardless of the device they are displayed on. Elements with text inside are sized according to the size of their text to ensure they always display correctly. To help those who require or prefer a larger font size the setting menu offers an option for a larger font to make the text bigger everywhere throughout the app.

6.7 Iterative Development Process of the Settings

The settings are available from the top action bar by tapping the cog wheel icon. Presenting settings to the user in an intuitive way while still providing a lot of user control on a small screen was more difficult than it would have first seemed. An iterative approach was taken where additional settings were added as the app was developed and the settings page was redesigned several times. It was decided to keep all the settings on a single page to reduce the navigation complexity and avoid options being hidden by being buried deep in sub-settings. The challenge then was to place all the settings on the narrow space available on most mobile devices so that scrolling up and down the page it would be easy to understand what they do.

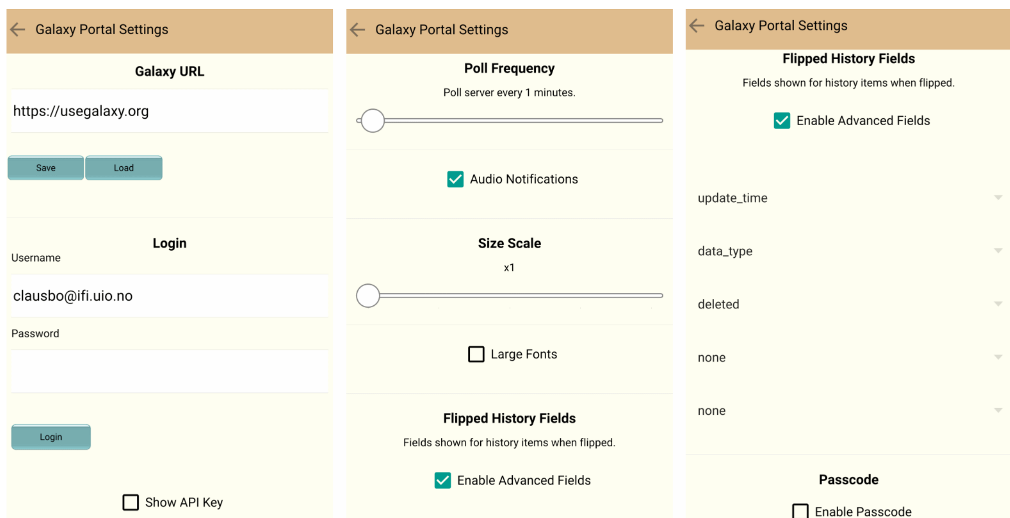


Figure 16: The settings are all on a single page, but on a mobile device the user can only see a segment of the page at any time as shown here from an Android mobile phone.

When optional settings such as API key, advanced fields and passcode are not enabled the associated fields are hidden - collapsing the settings menu and so making it easier to navigate.

The code structure behind the settings page started out as a single file, but as more settings were added the code for each part of the page was refactored into its own file with the settings page simply handling the overall drawing of the page using these components. This modular approach made it easy to maintain, expand and restructure the settings page as needed.

The only sub-menu is the Galaxy instance list which brings up a list of all saved Galaxy

connections and allows loading a previous instance or deleting the connection by tapping the trash can.

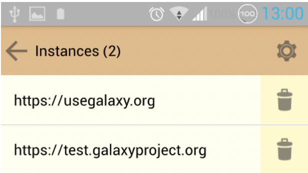


Figure 17: Two Galaxy instances that have been saved. Connecting to an instance is done by simply clicking on it in the list.

6.7.1 Login

In the initial version of Galaxy Portal the connection could only be established by typing or pasting the Galaxy API Key from the Galaxy system into the API Key field. During testing this proved cumbersome, and login using username and password was suggested by one of the Galaxy team members after testing the app. This was added using the Galaxy API module which allows logging in and retrieving the Galaxy API Key using basic access authentication. Since the Galaxy system generates an API Key if the user does not have one this proved to be a far easier method for gaining access and the settings page was redesigned to make this the primary method of connecting.

6.7.2 Touch-Based Paste Functionality

To make it possible to paste into the url, username and API Key fields on all types of devices it was necessary to add a paste icon and code that pastes the content of the clipboard into the highlighted field. The paste icon is an action bar button that only appears when an appropriate field is selected. The need for this functionality was only discovered during testing just like other important features such as controlling the poll frequency, audio notifications, size scaling and of course logging in with username and password.

6.7.3 Sliders

The poll frequency and scale multiplier are both set with a Qt Slider. The slider increments and decrements a value by a fixed step size as would be expected. The only really interesting part is that the text element above the slider shows the value by combining a static text string with the value set by the slider.

```
text: pollFrequencyField.value === 0 ? qsTr("No update polling") :
```

```
qsTr("Poll server every ") + pollFrequencyField.value +
qsTr(" minutes.")
```

Unlike when assigning a static string that will not change, setting the slider value property creates a one-way property binding from that property to the text element so that any time the value changes the text element is updated. The Qt framework supports such one-way property bindings and ensures they always stay in sync so that any change to the value will be reflected in all properties bound to it. However, Qt does not support bi-directional property bindings where two properties bind to each other and so a change to either will update the other. This can be achieved with two one-way bindings, but additional code is then required to check that the value has actually changed before updating to avoid a binding loop.

6.8 Event Thread and Background Thread

Having the app notify the user when the status of a job changes emerged during testing as an important feature. To save the user from having to take out the phone and visually check status an audio notification is played when it is detected that the status has changed during the model update triggered by the periodic server polling. A different audio notification is used when a running job completes, or encounters an error, to make it possible for the user to learn to recognise what is happening without having to actually check the status on the device.

For this feature to be useful it was important that status updates could be detected while the app was running in the background or the device has gone into standby mode. Qt Quick has a `WorkerScript` QML type which runs in its own thread, but on implementing this solution it was discovered that there was no option for putting the thread to sleep using QML or JavaScript. The `WorkerScript` thread is intended for performing operations without blocking the main user interface event thread, but without a sleep function that could block the thread from taking up resources at intervals it was not suitable for running as a permanent background thread.

Instead a separate thread was implemented in C++ to run independently of the user interface event thread. The Qt `QThread` class was used to implement a Qt style platform independent thread that could take advantage of the signals and slot system to send and receive signals from the main user interface event thread. A `Ticker` class inheriting from the `QObject` base class is set as the worker class by using the

`QObject::moveToThread(QThread * targetThread)` method to set it running in the separate `QThread`.

```
QThread* thread = new QThread;
Ticker* ticker = new Ticker();
ticker->moveToThread(thread);
```

6.8.1 Signals and Slots

This C++ thread cannot communicate directly with the main event thread in the `QGuiApplication` object and so a separate C++ class inheriting from QObject named `Bridge` is instantiated to act as a signal bridge between the two threads. A number of signals and slots are then set up between the `Ticker`, `Bridge` and main `QGuiApplication` event thread. We will not look at all these here, but the interested reader can find the functions for the signals emitted and slots for receiving signals in the `Ticker` and `Bridge` classes, and the corresponding `QObject::connect` calls to connect the signals to slots in the `main.cpp` main function, such as shown here.

```
QObject::connect(ticker, SIGNAL(tick()), bridge,
            SLOT(tick()), Qt::BlockingQueuedConnection);
```

Note that this connection is setup as blocking to ensure communication is synchronous between threads, but this may not be necessary in this case since we are just sending a tick signal once in a while - every minute for the most frequent polling interval. The reason to remove it is that a blocking connection will not return until the call has been processed and so could potentially lead to some inefficiency and in a more complicated scenario blocking calls can potentially also lead to deadlock. Neither of these are real problems here, but as more app functionality is added and the background thread potentially takes on more functionality this could become an issue.

One other interesting thing to note is that the polling frequency is set from the settings page in the QML user interface, and propagated back to the `Ticker` thread by calling the `setTickInterval` slot in `onPollIntervalChanged` in the `JSONListModel` QML component. An observant reader will note that there is no `QObject::connect` call to this slot, but instead the signal and slot connections between the event thread and `Bridge`

class are setup automatically by binding the `Bridge` class to the `QQuickView` context as shown here.

```
viewer.rootContext()->setContextProperty("Bridge", bridge);
```

The thread is simply a while loop in the `mainThread` function of the `Ticker` class. This function is a slot connected to the `QThread::Started()` signal emitted when the thread starts execution and which runs for the duration of the thread, which ends when the `QCoreApplication::aboutToQuit()` signal is emitted when the main event loop is about to quit and thus the app is about to terminate.

6.8.2 Background Processing on Mobile Devices

The idea was initially to have the thread perform the periodic server polling and keep a local cache so it could detect status changes and trigger the notification, and would then only need to send the delta (i.e. any variation in the data model) of any updates to the QML list models. However, as it turned out there is no Qt C++ equivalent of `XMLHttpRequest` and writing a new class would have taken too long and seemed redundant when there is a perfectly good JavaScript method.

At the same time it was discovered, despite various online community forums suggesting otherwise, that the separate C++ thread also stopped running when the app went into the background or the device went into standby. The solution it turned out was to enable the app to run in the background in the Android xml manifest file deployed with the app by specifying a meta-data parameter for this, as shown below.

```
<meta-data android:name="android.app.background_running"
android:value="true"/>
```

This not only keeps the C++ thread running, but prevents the app from being suspended and so also keeps the main event thread running. This means that we can use signals and slots to call the `XMLHttpRequest` JavaScript method from the C++ background thread and update the QML model directly. Even the audio notification is a QML audio element which plays when the QML model detects a status change, and so we could replace the C++ thread with a QML timer element. This may indeed end up as the final solution, but at the moment the C++ thread is kept in case it turns out that it will still be useful if background mode is implemented for iOS.

In iOS a background mode must be specified in the property list configuration xml file (Info.plist), but we must also specify a specific background mode. There are a number of very specific background modes, such as “audio” which prevents the app from being suspended while it is playing audio or video. The desired background mode for the Galaxy Portal app would be “fetch” which causes the system to wake the app up when system resources allow to poll for data. This is specified in Info.plist as shown below.

```
<key>UIBackgroundModes</key>
<array>
    <string>fetch</string>
</array>
```

Unfortunately, when the app is woken the system calls a delegate handler for which support has not been implemented in Qt as indicated by the unresolved QTBUG-42934[69] reported 27th of November 2014. Because of this the app currently only runs in the background on the Android platform. Since Qt is open source it would be possible to add iOS support for `UIBackgroundModes` in Qt, but this has not been attempted due to time constraints.

69 See https://bugreports.qt.io/browse/QTBUG-42934; accessed March 30 2015

Part III

Dissemination

Chapter 7

Distribution

The open source version of the app was made available to download for Windows and Linux by providing the release binaries on the GitHub repository, and for Android through the Google Play marketplace. However, to distribute the app on the iTunes app store for iOS devices the app had to be dual licensed which meant investigating the legal options and costs involved. We shall first look at the various versions released and on which platform it was distributed and then we will take a look at the iOS and GPL conflict which made dual licensing the app necessary.

7.1 Releases

The development project moved through three stages from early alpha and beta versions available only to invited testers on Android and iOS, to release versions freely available to anyone to install from the iOS or Android marketplace, or download as binaries or source from the GitHub source repository. The distinction between alpha and beta was only intended to signal the level of completion.

The binaries for each version released were made available together with corresponding archived source on the GitHub repository, along with instructions for how to install and changes in each version. There were eleven releases in total, although the final 1.1 version was only updated on the iOS and Android marketplace as it contained only specific fixes for those platforms. Version 0.1 and 0.2 were never released and only used for internal testing, and version 0.8 never went live, and so these are not included here.

Version	Date	Supported Platforms
0.3 (alpha)	28/10/14	Windows
0.4 (alpha)	02/11/14	Android, Windows
0.5 (alpha)	09/11/14	Android, Windows
0.6 (alpha)	18/11/14	Android, Linux, Windows, (iOS)
0.7 (beta)	10/01/15	Android, Linux, Windows, (iOS)
0.9 (beta)	09/02/15	Android, Linux, Windows, (iOS)
1.0 (release)	20/03/15	Android, Linux, Windows, iOS
1.1 (release)	25/03/15	Android, Linux, Windows, iOS

Table 3: Versions released for Galaxy Portal with dates and platforms supported for each version. iOS was only available to testers outside the team from release 1.

7.2 iOS and GPL Conflict

When Qt was first considered as the cross-platform framework the GPL license was not seen as a risk. The project was planned as an open source project from the start and the GPL license seemed like a great choice for the sort of scientific app planned. The license of the Galaxy system is not an issue as the app is only communicating with the API and the GPL license does not specify or restrict the operating system it can be used with in any obvious way.

Other open source licenses were considered, such as the MIT and BSD, but although source licensed under those open source licenses are compatible for use with the GPL the app would still need to be released under the GPL due to using the Qt framework.

The other alternative that was considered was to take advantage of the Qt LGPL license. This would allow the app source to be linked to the Qt libraries and so could be released under pretty much any license. This would have been useful if there was a desire to keep the app source closed, but as the app was intended to be open source software there seemed little point. Furthermore, while looking into this possibility it was discovered that iOS might not allow dynamic linking of libraries and so the libraries would need to be statically linked. According to the GNU operating system frequently asked questions about licenses[70] a statically linked app would also need to provide the app in object format (or source) to give the user an opportunity to modify the library and re link to the app. With the latest version of iOS available there were indications that it may be possible to dynamically link libraries, but it also seems that two of the terms in the App Store

70 See http://www.gnu.org/licenses/gpl-faq.html#LGPLStaticVsDynamic; accessed March 25 2015

Review Guidelines[71] for developers prohibit this in apps:

2.7 Apps that download code in any way or form will be rejected
2.8 Apps that install or launch other executable code will be rejected

Thus it may be technically possible to dynamically link libraries, but the option might still be blocked legally. This option was not explored further as using the LGPL did not seem to offer any benefit over using the GPL for the intended purpose of developing the app as open source.

Another aspect of the development process was the desire to use the Git[72] source control system with the GitHub source repository. Only open source code can be stored on GitHub without paying for a private repository, so again the GPL seemed like a great fit. It was also noted that that using the GPL would require anyone deciding to use a branch of the source for their own project – typically referred to as creating a "fork" - to also release their source with the same license under the terms of the GPL. So while the GPL does not in any way prevent commercial use it does prevent any use without using the same license and sharing the source along with any changes made, such as new functionality added. Although less restrictive licenses like the MIT, BSD or Apache license create fewer legal complications and are less likely to get developers into trouble they also do not guarantee that forks of the source will remain open source. Anyone can take source under such a license and develop it as closed source commercial software and make a profit from it without sharing any of the work. In contrast the sharing enforced by the GPL supports the vision of a scientific app that is developed for the benefit of the community.

7.2.1 App Store Terms and Conditions

The unexpected downside to the GPL in a cross-platform app was only discovered when doing research on how to build and deploy the Qt app to iOS. While learning about this process warnings were seen on forums and blogs that apps distributed under the GPL were not allowed on iOS. Further research revealed that the problem was not the iOS OS itself, but rather the terms and conditions that all users must agree to when using apps from the App Store as shown by this introduction that governs use of among other things iTunes Store, Mac App Store and App Store: "THE LEGAL AGREEMENTS SET OUT BELOW GOVERN YOUR USE OF THE ITUNES STORE, MAC APP STORE, APP STORE, AND IBOOKS

71 See https://developer.apple.com/app-store/review/guidelines; accessed January 25 2015
72 See http://git-scm.com; accessed March 25 2015

STORE SERVICES ("SERVICES"). TO AGREE TO THESE TERMS, CLICK "AGREE." IF YOU DO NOT AGREE TO THESE TERMS, DO NOT CLICK "AGREE," AND DO NOT USE THE SERVICES."[73]

Determining if these terms and conditions conflict with the GPL in the legal sense is not easy and requires legal expertise, and even then may not be determined with absolute certainty. The strongest indication of a license conflict is a number of GPL applications that have been removed from the Apple App Store such as the GPL licensed VLC media player and the GNU Go game. The FSF talked about their App Store GPL enforcement of GNU Go at the time and pointed out that the App Store Usage Rules imposed additional terms and required users to agree to stay compliant with these usage rules whether or not an app enforces this[74]. This involved version 2 of the GPL, referred to as GPLv2, and violated section 6 (section 10 in GPLv3) of that license which had the sentence "You may not impose any further restrictions on the recipients' exercise of the rights granted herein."[75]

The wording of the terms have changed somewhat since then and the GPL version used by Qt is version 3, referred to as GPLv3. There are no known cases of GPLv3 distributed on the App Store. The general consensus among developers on forums and blogs seems to be that Apple do not currently accept GPL licensed apps, but again this has not been confirmed either way. The fact that the same document discusses terms and conditions for iTunes Store, Mac App Store, App Store and IBooks Store makes it hard without careful study to understand what terms applies to iOS apps in particular. However, the section on "APP STORE PRODUCT USAGE RULES" for example looks like it conflicts with the requirements of the GPL: "(i) If you are an individual acting in your personal capacity, you may download and sync an App Store Product for personal, non-commercial use on any iOS Device you own or control."[76] This seems to restrict the usage of the app which would violate the intent of the GPL. However, without legal counsel it is difficult to assert with any certainty if this or other parts of the terms and conditions are in conflict with the GPLv3. Another point is that the very fact that these terms and conditions are updated from time to time creates a running risk of conflict with the requirements of the GPL.

It is also interesting to note that the iOS Developer Program License Agreement specifically mentions FOSS software and the GPL, but only to the extent of assigning all responsibility to the developer for any licensing terms governing the application and in par-

73 See http://www.apple.com/legal/internet-services/itunes/us/terms.html; accessed March 13 2015
74 See http://www.fsf.org/blogs/licensing/more-about-the-app-store-gpl-enforcement; accessed November 13 2014
75 See http://www.gnu.org/licenses/gpl-2.0.html; accessed March 13 2015
76 See http://www.apple.com/legal/internet-services/itunes/us/terms.html; accessed March 13 2015

ticular for matters related to digital signing and content protection.

In view of the above it does not seem advisable to release the app on iOS using the GPL. Even if the app was approved by the Apple review there would be a strong risk that it could run into legal problems from users, or potentially Digia Plc. This can be especially serious since failing to fully comply with the GPL license can result in the right to distribute the software being revoked. We shall not delve deeper into the intricate legalities of this here, but simply note that a developer with limited resources must be very careful when distributing software licensed under the GPL.

7.2.2 Dual Licensing

One possible solution is to release the app under a different license. Software can be released under multiple licenses as long as everyone with a copyright, i.e. all involved contributors, agree.

Qt itself is a perfect example of this. Released by Digia Plc in different version under GPL, LGPL and three commercial licenses: Indie Mobile, Professional and Enterprise[77]. The indie mobile license from Digia Plc is a recently introduced license which appeared towards the end of this thesis project, and a perfect fit for dual licensing Galaxy Portal. As the sole developer of Galaxy Portal, at this stage, I can easily change the license on my code. However, I would still need to comply with the license from Digia Plc and thus a commercial license, like the indie mobile license, would need to be obtained for the Qt framework. This would allow releasing the app under separate licenses.

To do this two separate sets of source would have to be maintained. The commercial version of the Qt framework must be downloaded and built separately and the app source would have to be compiled with this version to avoid the GPL. Obviously it is then very important to ensure there is no GPL licensed code that I as the sole developer cannot dual license, but since as discussed the other licenses are all BSD and MIT licenses there should be no problem in this regard.

To establish my options and confirm my understanding of releasing Galaxy Portal with multiple licenses Qt Support was contacted. A few emails were exchanged, but it emerged that it was a sales representative who was not able to advise on legal issues and so on the license I would need. What was provided was a quote for a Qt Enterprise license for a single developer to release on iOS for 3195 euros, and when that was declined it was confirmed that the Indie Mobile license would be an option for a short term project stressing that I would not have access to Qt technical support and cloud services.

77 See http://www.qt.io/download; accessed March 10 2015

The only licensing advise I was offered; "If you start the development with the LGPL license, you need to release the commercial product under the same license", did not seem correct.

I therefore turned to my lecturers from the “Open Source, Open Collaboration and Innovation” (INF5780) course I took the year before asking them if I as the owner of my code could distribute it under multiple licenses as long as I used a corresponding license for the Qt framework. In other words, if if I purchased a commercial license of Qt could I then distribute a commercial version of the app while continuing to distribute the GPL version. To illustrate this I included the figure shown below in my email to them.

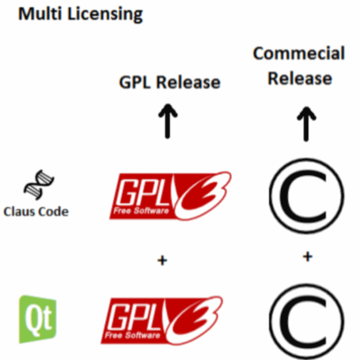


Figure 18: Original attempt to illustrate releasing of the Galaxy Portal app under multiple licenses. Starting from the bottom the image shows two parallel paths of licensing. One with Qt and my source using GPLv3 for an open source GPL release, and another with Qt and my source using commercial licenses for a commercial release aimed at the iOS market.

I was lucky enough to get a detailed response not only from my lecturer Wolfgang Leister, but he was kind enough to also send my query onwards to his colleague Trenton Schulz who it so happened had previously worked for Qt. They pointed out that I could have multiple product lines as long as the licenses chosen for Qt and my source are compatible. In particular my intended choice of dual licensing my source with a GPL license and a commercial license was valid. For Qt it was suggested to use LGPL for all platforms except iOS where a commercial version of Qt was suggested used together with a commercially licensed version of the app source or using the Mozilla Public License (MPL)[78]. It was of course pointed out that I would need to be careful to link with the correctly licensed version of the Qt libraries. I have tried to accurately reflect what they advised,

78 See https://www.mozilla.org/MPL/2.0; accessed April 2 2015

but if inaccurate this is doubtlessly because I have inadequately interpreted what was advised to me.

As the app was already released under the GPL license that version was maintained under the same license for distribution on all platforms except iOS. A commercial license for Qt was obtained which allowed the commercial version of the Qt framework source to be downloaded to a Mac. A copy of my own source licensed under a standard copyright was then modified to reflect the new license, built, packaged and deployed for review on the iTunes marketplace. After eight days it was approved and so made available for installing on iOS devices[79].

79 See https://itunes.apple.com/us/app/galaxy-portal/id937158087; accessed April 12 2015

Chapter 8

Writing for a Scientific Journal

While working on this thesis the idea of writing a journal paper was suggested as an interesting additional goal for the thesis, and a great way to reach the target audience for the Galaxy Portal app. To make it more interesting in the context of master work it was decided that I would first author an initial draft on my own and then additional co-authors would be invited to collaborate on revising it into the final version for submission.

The paper was written as a short manuscript to be submitted as an Application Note to the journal Bioinformatics from Oxford University Press which focuses on genome bioinformatics and computational biology. As these are emerging fields that rely heavily on software tools to analyse data the journal has a section dedicated to Application Notes - a short technical paper to communicate new software, algorithms and other services relevant to the readers of a scientific journal. This format and journal was identified early on as the best choice for a scientific paper on Galaxy Portal.

8.1 The Writing of a Journal Paper

To write the paper a number of existing application notes were read and the submission guidelines studied. All the application notes were around two pages. This is the maximum for an application note, but to increase the odds that the paper would be accepted and keep the paper focused it was decided to aim for only a single page. Much of the available page space is taken up by the header, and practical details about the software's availability and implementation (it must be freely available to qualify), as well as references.[80]

80 For the full submission guidelines see http://www.oxfordjournals.org/our_journals/bioinformatics/for_authors/general.html; accessed March 14 2015

With such a short format to present the app the experience was very different from writing this thesis, and it was especially challenging to determine what would be interesting and relevant to the readers and presenting it in the very limited space available. Much of the challenge in writing this thesis has been to organise the structure so that each part builds on previous chapters, and to achieve a balance between not boring the reader with too much repetition, while still making sure a connection can easily be made with earlier material. In contrast the journal paper was so short that the challenge was to write a concise text that would present to the reader not only what the Galaxy Portal app does, but why this is useful and how it was done.

In writing the journal paper I was forced to put myself in the shoes of the researcher and think how they would use the app and what benefit it would bring them. While understanding user requirements is an important aspect of software development the focus tends to be on the best technical solutions. It is easy to lose sight of the original purpose of the software being developed in the quest for clever code and neat features, but when writing such a short paper you are forced to re-examine what is the most important core functionality and discard all the rest.

The first draft of the paper, written solely by me without any input, attempted to briefly present the major features of the app, but during the later versions - written in collaboration - the app as a monitoring tool became the focus with more emphasis on the benefits this can bring.

8.2 Original Draft

Presented here is the original version drafted by me as the sole author and without any assistance or editing help. The intention was both to create a work that could be presented in the thesis as mine, and as a starting point for collaboration on a revised version to be submitted. This original draft is presented on the next page as it was originally formatted with only minor modifications to fit on a page of this thesis (which is slightly smaller than the template format used by the Bioinformatics journal).

Sequence Analysis

Galaxy Portal: Easy Access to Galaxy Biomedical Research Data

Claus Børnich[1,*], Co-author[1] and Co-Author[2]

[1]Biomedical Informatics, Department of Informatics, University of Oslo, Norway.

[2]Department of XXXXXXX, Address XXXX etc.

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

Summary: Galaxy Portal is an open source project to provide a modern user interface for smartphones and tablets to the Galaxy bioinformatics analysis platform.

Online access is increasingly done through mobile devices, such as smartphones and tablets. Their portability makes it convenient, quick and easy to get online from almost anywhere at any time. Using a browser for web access on touch-based devices can be a slow, difficult and frustrating experience, and the Galaxy user interface was not designed for small touch screens with a relatively low precision compared to using a mouse. The Galaxy Portal app therefore aims to make accessing Galaxy on mobile devices easy and fast, while presenting data tailored for a small screen and supporting features such as touch screen interaction, and device orientation that allows the user to change from landscape to portrait mode by simply rotating the device.

Availability and implementation: The source is freely available under a GPL license on Github, along with user documentation and pre-compiled binaries and instructions for several platforms: https://github.com/Tarostar/QMLGalaxyPortal. The iOS version is available through the Apple App Store for iOS7 and newer, and Android through Google Play for version 4.1 (API 16) or newer. The Linux and Windows releases should in theory run on all modern versions of the platforms, and have been tested on Linux Ubuntu and Windows 8.1.

Contact: clausbo@ifi.uio.no

1 INTRODUCTION

Galaxy (Goecks *et al.*, 2010) is a widely used open source, web-based platform that runs automatic analyses jobs for data intensive biomedical research. Researchers often need to run a number of such jobs that can take days to complete. While the current version of the Galaxy Portal app cannot be used to run new jobs it does provide an easy to navigate, responsive and low-bandwidth interface to monitor the status and take a peek at the results and dataset metadata. This gives researchers a convenient and immediate means to monitor when large analysis tasks complete or encounter errors from their mobile phone, tablet or laptop, and so can help improve productivity.

Furthermore, Galaxy tracks a user's execution history in a reproducible manner and the Galaxy Portal app provides a fast and simple way to browse that history.

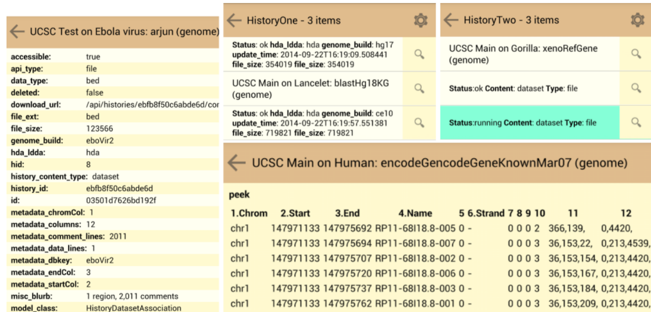


Fig. 1. User interface on Android showing available information in the dataset and status colour of a running item in a history list of three items.

2 FEATURES and METHODS

Qt (http://www.qt.io), a cross platform framework, has been used to develop an app that compiles to an OS native app, and provides a touch based interface that scales according to screen size and resolution. The Qt framework is C++ based, but the app was primarily developed using JavaScript and the Qt Modelling Language (QML). Compiling to a native Android and iOS app ensures that it performs optimally, while being able to implement the user interface and logic using QML and JavaScript allows for rapid development of a sophisticated and responsive interface using animations and platform native styling.

Access to Galaxy data is achieved using the RESTful Galaxy API (Sloggett *et al.*, 2013) to retrieve JSON formatted data which keeps bandwidth use low, and the polling frequency can be configured by the user. Interfacing to Galaxy through the API also means that the app can connect to any Galaxy site, or any tools built on top of the Galaxy source. The app allows researchers to keep a list of connections to enable easy switching between sites.

Future versions of the app could be extended to allow jobs to be scheduled providing a more complete mobile interface for Galaxy.

Conflict of interest: none declared.

References

Goecks,J. *et al*. (2010) Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. , 11,R86.

Sloggett,C. *et al*. (2013) BioBlend: automating pipeline analyses within Galaxy and CloudMan. Bioinformatics, 29, 1685–1686.

8.3 Paper Writing Process

The writing process started with me writing a complete first draft. I used a template with the correct layout, and as can be seen on the previous page the draft consisted of an abstract followed by the two sections "Introduction" and "Features and Methods", as well as references and a figure showing four screenshots of the Galaxy Portal app.

This draft was then reworked by supervisor and two more faculty members invited to join in the process. Most of the collaboration was done online, through emails and google documents, but two meetings were held. These served as brainstorming sessions and helped to establish a common focus for the application note. As a consequence there was a shift to focus on the monitoring utility of the app as its most important feature. Additional ideas for the app were also discussed. It was identified that being able to tweak and then re-run jobs would have been a very appropriate feature, but unfortunately the amount of work involved was beyond what could be achieved in the time left and is therefore instead something to consider for future versions. However, the audio notifications to push alerts to the user about status change suggested by one of the co-authors was subsequently added as a new feature.

After several rounds three of the members on the Galaxy team in the US were invited to participate. All were interested in participating, but only one of them was able to provide feedback in the time available. In fact, most of the authors had a busy schedule and soliciting feedback and arranging meetings took up several weeks for each round of revisions.

8.4 Changes From Collaboration

The original had a long abstract which presented Galaxy Portal as a modern user interface for Galaxy, and then presented the case for a mobile user interface, along with its features. The idea was to promote the benefits of having a mobile specific app interface, but with hindsight it is easy to see that this resulted in a lot of focus on trivially obvious features such as touch interaction and adapting to screen orientation.

As a result of the first brainstorming phase of the revision a lot of additional background material was added to the abstract[81], but then in subsequent rounds most of it was cut. This was necessary to stay within one page, as well as the submission guidelines for the abstract of around two sentences, and 160 words. Most of this was achieved in an intensive writing session with three of the authors that resulted in the abstract in the ver-

81 See appendix C.1.

sion at the end of this chapter, with a tight focus on core functionality and why it should be considered useful.

The introduction chapter was also extended to be longer than in the original, as it was filled out to be much more thorough in arguing for the benefit of the app to make monitoring more convenient and efficient, and with an emphasis on the ease of use and other features also mentioned in the original version - such as low bandwidth use. The final paragraph of the introduction was also expanded into two longer sentences with more careful wording to explain Galaxy Portal's ability to browse Galaxy histories and why this is useful. This deviates slightly from the focus on the app as a monitoring tool, but again provides a strong case for why the feature is useful.

The final section on features and methods stayed essentially the same in content except for a sentence on the passcode feature and not storing the password, but was significantly restructured and rephrased.

On suggestion from some of the co-authors I added a mobile frame to the screenshots to make it easier to understand the size and layout of the Galaxy Portal app. The initial version used an older iPhone frame which resulted in only three images of the app and when this new version was shared with the US co-authors the immediate feedback was to remove the frames to be able to show more images. As a compromise I again reworked the images taking care to choose a thin frame. The figure text was also updated to be shorter and more direct.

As can be seen below in section 8.5, the final version benefited from the experience of the additional collaborators and several eyes participating in the writing and editing process. The use of the Latin "*á priori*" and other changes to the wording, grammar, style and content produced a much more professional manuscript which should have a much better chance of being accepted for publication. In short, the collaboration resulted in a more considered focus, improved language and better arguments for the use of the app with more background knowledge than I as the original author could have supplied.

8.5 Final Version

Presented here is the final version as a result of collaboration with Eivind Hovig, Jonas Paulsen and Geir Kjetil Sandve from the University faculty and Martin Čech from the Galaxy team.

Genome Analysis

Galaxy Portal: Monitoring Galaxy job execution on mobile devices

Claus Børnich[1,*], Eivind Hovig[1,2,3] , Jonas Paulsen[4], Martin Čech[5] and Geir Kjetil Sandve[1]

[1]Biomedicial Informatics, Department of Informatics, University of Oslo, Norway, [2]Department of Tumor Biology, Institute of Cancer Research, [3]Department of Cancer Genetics and Informatics, Radium Hospital, part of Oslo University Hospital, Montebello, Norway, [4]Institute of Basic Medical Sciences, University of Oslo, P.O. Box 1112 Blindern, 0317 Oslo, Norway, [5]Department of Biochemistry and Molecular Biology, Penn State University, USA.

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

Summary: We here present Galaxy Portal, an open source interface to the Galaxy system through smartphones and tablets. The Galaxy Portal app provides convenient and efficient monitoring of job completion, as well as opportunities for inspection of results and execution history. In addition to being useful to the Galaxy community as it is, we believe that the app also exemplifies a useful way of exploiting mobile interfaces for research/high-performance computing resources in general.

Availability and implementation: The source is freely available under a GPL license on GitHub along with user documentation, pre-compiled binaries and instructions for several platforms: https://github.com/Tarostar/QMLGalaxyPortal. It is available for iOS version 7 and newer through the Apple App Store, and for Android through Google Play for version 4.1 (API 16) or newer.

Contact: clausbo@ifi.uio.no, geirksa@ifi.uio.no

1 INTRODUCTION

Modern biomedical research often depends on compute-intensive data processing and analysis at different stages of an investigation. Galaxy (Goecks *et al*., 2010) is a widely used platform for genome analysis, providing a web-based interface to initiate and monitor computations executed on a server. A biomedical investigation may often involve several such computations running for a long and *á priori* undetermined time period. Often, projects depend on such computational results for further progress. This may lead researchers to constantly go back and forth to a computer to monitor job execution progress. Access through mobile devices could make such monitoring much more convenient and efficient.

Galaxy Portal is an open source interface to the Galaxy system through smartphones and tablets. Rather than supporting the full functionality of Galaxy, the Galaxy Portal app limits itself to monitoring job completion and inspection of results. The purpose is to provide a simple and efficient interface that can be accessed from almost anywhere at any time through a mobile device. The app is responsive and low-bandwidth with data tailored for a small screen.

Furthermore, Galaxy tracks a user's execution history in a reproducible manner, and the Galaxy Portal app provides a fast and simple way to browse the Galaxy history elements. This can be very useful as a way of quickly checking results or details of exactly how the given results were produced, for instance in a discussion setting where a mobile device is more convenient to use than a computer.

Fig. 1. User interface on Android, showing left: histories, middle: jobs with a running item, and right: landscape oriented examples.

2 FEATURES and METHODS

The app was developed with the Qt cross-platform framework (http://www.qt.io) which compiles to native iOS and Android apps for optimal performance. The Qt framework is C++ based, but the app was primarily developed using JavaScript and the Qt Modelling Language (QML).

Access to Galaxy data is achieved using the RESTful Galaxy API (Sloggett *et al.*, 2013) to retrieve JSON formatted data, which keeps bandwidth use low. Due to the use of the API, the app can connect to any Galaxy site, or any tool built on top of the Galaxy source. The app allows researchers to keep a list of connections to Galaxy instances, to enable easy switching between sites.

The app provides a touch-based interface that scales according to screen size and resolution. Access to the app can be passcode protected and the user' password is never stored.

Conflict of interest: none declared.

References

Goecks,J. *et al.* (2010) Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. , 11,R86.

Sloggett,C. *et al.* (2013) BioBlend: automating pipeline analyses within Galaxy and CloudMan. Bioinformatics, 29, 1685-1686

Part IV

Discussion

Chapter 9

Final Remarks and Further Work

This has been a very interesting Master thesis which has taught me much about not only app development for mobile platforms and the Qt framework, but also served as a real test case for learning about licensing and given me a taste of scientific publishing. I have been able to learn and test multiple development frameworks and build and deploy software to several operating systems, as well as learning how to interface with the RESTful API the Galaxy system provides.

I feel that I have come a long way, but I also think that armed with a bit more experience I could have gotten much further. I sincerely think that this app has great potential as an alternative Galaxy interface that could become incredibly useful with the implementation of additional features such as the ability to submit jobs. I therefore hope that this thesis will inspire other master students to continue the work and that this document together with the code will make it possible to turn Galaxy Portal into a widely used and highly useful tool for users of the Galaxy system.

In this final chapter I would like to summarise the diverse work done in this thesis and make a few remarks on the lessons learned and where we can go from here.

9.1 The Licensing Challenge

When planning a project, considerable time and effort may be spent on identifying and mitigating against perceived risks, but the more unlikely the risk the less likely we are to discover it – let alone spend time planning for it, and yet it is exactly such unseen obstacles that pose the greatest risk to a project. Licensing was most certainly seen as a risk, but that the choice of license could affect distribution on certain platforms was not anticipated.

Free and open source software (FOSS) has provided a lot of great software and a wealth of code written by a community of developers and available to all. A number of licenses based on copyright law are available to help a developer share work on his or her own terms, but this diversity and the legal complexity it creates is a potential minefield for a project. All licenses used in a project must be compatible and as we have seen each license comes with its own requirements and obligations that must be satisfied. It is fortunate that a copyright holder can license software under multiple licenses, but as soon as code under other licenses is incorporated, or multiple developers contribute to a FOSS project, it becomes much more difficult to do so. It should also be remarked that although the project can be released under additional licenses the copyright holder cannot withdraw source released under previous licenses, except for in specific instances when a license establishes this right under certain conditions - such as not fulfilling the requirements of the GPL. It is therefore essential that licensing options are properly investigated at the start of the project and the consequences of that choice are understood. Given the often difficult language, and frequently ambiguous and uncertain legal claims, this can be one of the greatest challenges a software developer faces. The consequences of choosing the wrong license can be profound. Often fundamental in determining if a project attracts support or competition, and whether the software can and will be used, and whether or not the involved developer(s) will receive credit or be able to earn a living from their work.

9.1.1 The Galaxy Portal License Situation

To distribute the Galaxy Portal app on the intended platforms using the Qt frameworks multiple licenses had to be used. This added additional work, cost and extra maintenance, as well as legal risk and complications from maintaining two sets of source with incompatible licenses. Furthermore, the Qt mobile license must be paid on a monthly basis and if this is terminated it appears that work would have to stop on the commercial version. The license cost could of course increase and it is also possible that Digia Plc could withdraw the license altogether leaving the commercial branch of the project stranded. These things are not entirely clear at this point and would need to be clarified and so adds further time spent on investigation and correspondence.

This all shows that while using an established framework can be a huge benefit to the development process, especially for cross-platform software, there is a cost. The obvious cost to a developer is learning the new framework, and potentially coding languages used by that framework, discovering and handling bugs and unexpected behaviour, as well as understanding how to build and deploy the source to various systems. However, the less

visible and often hard to understand cost is that of legal license complications and compatibility issues, and the restrictions on what can be done with the software and source. The GPL, for example, requires the developer to make source available, and is quite specific about how this must be done and for how long. If a developer fails to comply he, or she, could face legal action or have their rights to use the framework under the license revoked.

In the case of this project the choice of license prevented the app from being distributed for testing on one of the intended target platforms, and introduced delays and much time spent researching licenses and legal matters with few clear cut answers. In the end the solution to dual license the software introduces not only additional development time, but also unexpected ongoing licensing costs.

The technical challenges in using a new framework were many, but often well documented and overcome with a bit of hard work and ingenuity. The license issues, however, were obfuscated with divided opinions, and little official documentation or advise. Thus leaving developers to figure out the legal implications on their own. As most developers do not have much education or experience in law and licensing this is likely to be a much greater challenge for most of us than the technical challenges.

9.2 Cross-Platform Development

Being able to deploy software across operating systems and devices from a single uniform source can be incredibly important for maintainability, and the long term survival of an app. However, this inevitably means building on top of some form of interface or abstraction layer that can wrap or convert the source to the binaries required by the target platform, and that in turn means added complexity, and dependency on libraries and tools beyond the direct control of the software developer.

This last point is especially relevant for app development in the fast moving world of mobile platforms, where as we have seen the need for support for new platforms can emerge relatively quickly. This makes it very important that the frameworks are being actively developed, or they will quickly become a barrier to deploy on new platforms or take advantage of new features on existing platforms. All of the frameworks and tools we have looked at were open source and so it is entirely possible to extend the source to provide additional support, but the amount of work and understanding required to do so makes this unlikely to be a viable option for small projects. We saw this in chapter six when we discussed enabling the app to run in the background and discovered that the framework does not support this for the iOS platform. The functionality was well documented by Apple, but the knowledge and amount of work needed to implement it was

beyond the resources available.

Developing software aimed at cross-platform deployment also means being aware of support for various technologies on target platforms and despite using a cross-platform framework, custom code was still needed. Some of this is confined to the deployment phase where configuration files such as the Android manifest XML file or iOS configuration XML file were needed, but when implementing the ability to flip list items using OpenGL shaders the code written had to take into account differences in driver implementations. The desired effect worked fine when testing on Windows and Linux, but once deployed on an Android phone the vertex shader enabled list items were drawn as pink rectangles. The issue was eventually tracked down to imprecise floating point values written as integers in the shader code, i.e. set as "1" instead of "1.0". Research revealed that the strict interpretation of the OpenGL specification used by the Android phone drivers prevented the shaders from being calculated correctly, whereas the Nvidia drivers on my desktop computers were able to handle it. The solution, to ensure all float values were specified with a decimal value, was not difficult to accommodate in this case, but serves to highlight that the same code may behave differently on different platforms and devices. Testing is always an essential part of development, and for cross-platform development this means deploying to and testing all functionality across a wide range of devices, and so significantly increases the time used for testing.

The problem is not confined to code either, but also affected assets which had to be prepared in formats to satisfy the requirements for each platform and a range of devices with different resolutions. In particular Android and iOS each have their own requirements for icon sizes and satisfying both properly can be a significant job if an app has many graphical elements.

9.2.1 Cross-Platform Testing and Deployment

An easily overlooked aspect of cross-platform development is having access to sufficient devices, operating systems and distribution platforms to carry out deployment and testing on a wide range of platforms. We have touched upon this in previous chapters and as we have seen there are options for running virtual devices, but it is something which should still be planned for.

During the development of Galaxy Portal I had easy access to machines with Windows and Linux, as well as several Android devices, but only limited access to a Macintosh (OS X) and iOS devices and no Windows Phone. The result is that development and testing often focused on the available platforms with much fewer releases for iOS. A plan for testing which included obtaining at least one device for each target platform would have

resulted in more even testing across all platforms, and potentially a Windows Phone version.

Furthermore, the license conflict with the Apple terms and conditions restricted iOS deployment to only internal testing and since a separate memberships must be purchased for OS X and iOS, no version was attempted for OS X although it would have been technically possible.

Also, having to learn the procedures, terms and requirements for building and deploying the app on different platforms added a lot of work. This is an important consideration when deciding on the target platforms to support, as each additional platform adds a significant overlay of testing and time deploying the app.

9.2.2 Qt and Cross-Platform Development

Qt proved a great asset when developing on multiple platforms, as the same libraries and Qt Creator IDE was available for each platform. This meant that the same code could be downloaded from GitHub on Windows, Linux and Mac, and without any modification could be immediately compiled and tested. This flexibility was very useful when developing for multiple target platforms providing convenience and the confidence that the code could be built on any available machine without a lengthy setup or configuration.

Qt generally had very good documentation and an active forum, but there is still some confusion around deployment to mobile platforms – in particular iOS and Windows Phone. The level of support for Windows Phone improved significantly in the time Galaxy Portal was developed, native styles were added for Android, and there was noticeable improvements made to the documentation for iOS, but especially earlier in the project it was often hard to find platform specific technical solutions. Several times the final answer was only found in the online bug tracking system and not in the documentation.

This not only highlights the importance of an active and well supported framework, but of being aware of how issues are reported and tracked. The same can be said for simply paying attention to what the developers and community are discussing, and new features and changes in new versions. This may seem obvious, but it is easy as a developer on a project to forget that your work is built on a larger framework which is also developing.

Having now worked with Qt, as well as several other cross-platform frameworks, it is clear to me that these go a long way to make it easier to support multiple platforms, but do not fully solve all the problems.

9.3 Overall Outcome

I was able to develop and distribute a cross-platform app using the Qt framework, although not without difficulty. Looking at the resulting app I think that using the Qt framework was the right choice. However, it was fortunate that a new affordable license option for mobile devices was added for Qt during the project as otherwise obtaining a commercial license would not have been possible and the app would not have been available on one of today's largest app markets.

Despite all the effort to make the app available on the iOS platform, most of the feedback since the app was released has been from the Android version. Early feedback from the official Galaxy Biostar forum and from faculty members at the University of Oslo has been positive, but for the app to be widely used it will probably need continued development and support.

Writing for a scientific journal was a very different experience. Instead of working on my own solving technical problems I had to work with others to collectively author a concise text aimed at bioinformatics researchers. Everyone involved had busy schedules and so the greatest challenge was to coordinate meetings and solicit timely contributions. Although there were often differences in opinion and not everyone's suggestions could always be included there were no conflicts and a consensus was reached each time. It was highly education and with the experience of my co-authors I was able to write and submit a manuscript that should have a reasonable chance of being accepted, and so fulfils the second goal of this thesis.

9.4 Future Development

There is a lot of potential to develop the app further to provide additional functionality that will make the app even more useful. As has been mentioned, the entire source along with all binary releases and user documentation is available on GitHub, and the GPL license means that anyone can make a fork to continue development in any desired direction. Anyone wishing to continue work on the app are also welcome to contact me to be assigned as contributors to the original Galaxy Portal project on GitHub.

Perhaps the most obvious focus of future development would be to implement an interface to enable submitting jobs. A good starting point, and very useful feature for Galaxy Portal, would be the ability to re-submit a job that has completed or encountered errors. It would be necessary to re-submit it with modified parameters to make such a feature useful, so there would need to be an interface to allow changing the parameters. Ideally, a dynamic user interface would be drawn to allow modifying the individual para-

meters of a particular job, but a better starting point might be to add a command line interface to allow modifying the URI that is passed to the Galaxy API[82].

It would also be worth checking if the delegate handler in Qt (QTBUG-42934) has been implemented as discussed in chapter six, and an ambitious student with sufficient experience in C++ might consider investigating the feasibility of contributing code to implement this feature in Qt.

There is also plenty of scope to investigate the Galaxy API for other functionality that might be useful and work with potential users at the faculty and members of the Galaxy team to determine useful features. As discussed in 5.2 the intention of the Galaxy team is to transition the user interface away from the Python controllers serving pages of HTML generated by a template engine. Instead the Galaxy team wants to perform all interaction with the servers through the RESTful API, and so there might be plenty of scope and support for developing what will be a stand-alone cross-platform interface using the API.

In the last week of the project a button was added to the detail view to display the full dataset for a job, and to copy it to the clipboard for sending in an email or pasting into a document. Because there is a very large amount of data the dataset is divided into a number of pages that the user can navigate between. This functionality was added to see if it could be done and has not yet made it into a release. Being able to view the entire dataset result could prove a very useful feature and so it would be worthwhile exploring ways that this data can be better formatted, and perhaps downloaded in a PDF or other formats. Furthermore, the Galaxy API also has experimental support for the visualization tools that Galaxy offers, and so functionality to generate and view graphs and charts could also be added.

The user interface can also be translated into additional languages. This possibility was considered when Galaxy Portal was implemented and so all text strings displayed to the user have been wrapped in the `qsTr` function which marks those strings for translation using the Qt translation system. The steps involved are well documented in the official Qt Linguist Manual and will primarily consist of adding XML-formatted translation files with localized text strings.

82 I have confirmed that it is possible to run analysis jobs through the API, but the documentation in this area is lacking so it might be necessary to investigate the Galaxy Python source available in the Bitbucket repository (see the `create` function in tools.py).

Bibliography

1. Nair AS. Computational biology & bioinformatics: a gentle overview. Commun Comput Soc India. 2007;2:1–13.

2. Miranda M, Ferreira R, Souza CRB de, Filho FF, Singer L. An exploratory study of the adoption of mobile development platforms by software engineers. Proceedings of the 1st International Conference on Mobile Software Engineering and Systems. Hyderabad, India: ACM; 2014. p. 50–3.

3. Padley R. HTML5 – bridging the mobile platform gap: mobile technologies in scholarly communication. Ser J Ser Community. 2011 Nov 1;24(0):S32–9.

4. Hansen TE, Hourcade JP, Virbel M, Patali S, Serra T. PyMT: a post-WIMP multi-touch user interface toolkit. Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces. Banff, Alberta, Canada: ACM; 2009. p. 17–24.

5. Virbel M, Hansen TE, Lobunets O. Kivy-A Framework for Rapid Creation of Innovative User Interfaces. Mensch & Computer Workshopband. 2011. p. 69–73.

6. Ulloa R. Kivy: Interactive Applications in Python. Packt Publishing Ltd; 2013.

7. Blanchette J, Summerfield M. C++ GUI programming with Qt 4. Prentice Hall Professional; 2006.

8. Thelin J. Foundations of Qt development. Apress; 2007.

9. Ezust A, Ezust P. An Introduction to Design Patterns in C++ with Qt 4 (Bruce Perens Open Source). Prentice Hall PTR; 2006.

10. Goecks J, Nekrutenko A, Taylor J. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. 2010;11(8):R86.

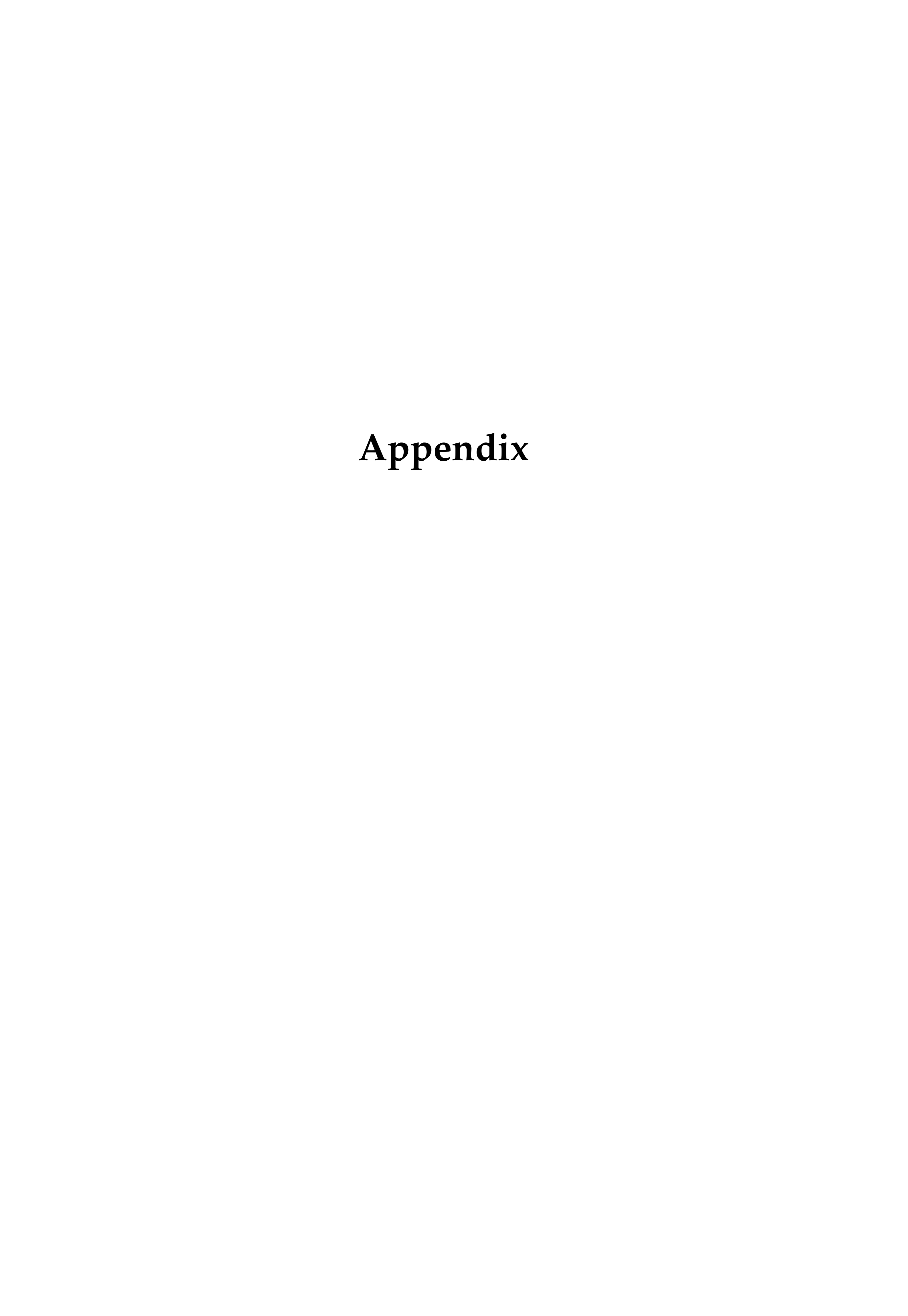
11. Richardson L, Ruby S. RESTful web services. O'Reilly Media, Inc.; 2008.

12. Fielding RT. Architectural styles and the design of network-based software architectures. University of California, Irvine; 2000.

13. Sloggett C, Goonasekera N, Afgan E. BioBlend: automating pipeline analyses within Galaxy and CloudMan. Bioinformatics. 2013;29(13):1685–6.

14. Leo S, Pireddu L, Cuccuru G, Lianas L, Soranzo N, Afgan E, et al. BioBlend. objects: metacomputing with Galaxy. Bioinformatics. 2014;Advance Access.

15. Sandve GK, Gundersen S, Johansen M, Glad IK, Gunathasan K, Holden L, et al. The Genomic HyperBrowser: an analysis web server for genome-scale data. Nucleic Acids Res. 2013 Jul 1;41(W1):W133–41.



Appendix

Appendix A

User Guide

The PDF guide was supplied to supervisor as A_GalaxyPortalUserGuide.pdf. It is also available online directly as a PDF from the GitHub source repository using this link: https://github.com/Tarostar/QMLGalaxyPortal/blob/master/GalaxyPortalUserGuide.pdf

Appendix B

Galaxy Portal Code and Binaries

The complete source was submitted to supervisor as a zip file: B_GalaxyPortal_CompleteSource.zip. It is also available online on the GitHub source repository along with all the binary releases (except iOS versions which are only available on iTunes). The whole source, along with the entire change history, can be browsed online, and the complete source can also be downloaded as a zip-archive by using the "Download ZIP" button provided on the main page using this link: https://github.com/Tarostar/QMLGalaxyPortal

Some code snippets referred to in the text are included here for convenience.

B.1 Utils.js - XMLHttpRequest Polling Function

```
function poll(source, onReady, parentID, authorizationHeader) {
    var request = new XMLHttpRequest;
    var pollTimer = Qt.createQmlObject("import QtQuick 2.3;
            Timer {interval: 5000; repeat: false;
                  running: true;}", parentID, "PollTimer");

    pollTimer.triggered.connect(function() {
                          request.abort(); onReady(); });

    request.open("GET", source);
    if (authorizationHeader !== undefined) {
        request.setRequestHeader("Authorization",
                              authorizationHeader);
    } else {
```

```
            request.setRequestHeader("Content-type",
                                     "application/json");
        }
        request.setRequestHeader('Accept-Language', 'en');
        request.onreadystatechange = function(){
                    pollTimer.stop(); onReady(request); };
        request.send();
    }
```

B.2 JSONListModel.qml - Updating JSON model

```
    // If we have json data - update the model.
    function updateJSONModel() {
          if (json === "")
                return;

          var objectArray = JSON.parse(json);
          // First remove any items from the model that no longer exist.
          for (var i = jsonModel.count - 1; i >= 0; i--) {
                var bFound = false;
                for ( var object in objectArray ) {
                    if (jsonModel.get(i).id === objectArray[object].id) {
                        if (objectArray[object].deleted === false) {
                            bFound = true;
                        }
                        break;
                    }
                }
                if (!bFound) {
                    jsonModel.remove(i);
                }
          }
          // Now update model and insert any new items.
          var index = 0;
          for ( object in objectArray ) {
                if (objectArray[object].deleted === true)
                {
                    // skip deleted item
```

```
                continue;
            }
            // Check if item exists at exactly the current index.
            if (index >= jsonModel.count) {
                // We are beyond the current model (or model empty).
                jsonModel.append(objectArray[object]);
            } else if (jsonModel.get(index).id ===
                              objectArray[object].id) {
                // Exists - check state and update.
                checkState(jsonModel.get(index).state,
                                    objectArray[object].state);
                jsonModel.set(index, objectArray[object]);
            } else {
                // Did not exist, see if at a different position.
                var bFound = false;
                for (i = 0; i < jsonModel.count; i++) {
                    if (jsonModel.get(i).id ===
                                    objectArray[object].id) {
                        // Found it, move it, check state and update.
                        bFound = true;
                        jsonModel.move(i, index, 1);
                        checkState(jsonModel.get(index).state,
                                    objectArray[object].state);
                        jsonModel.set(index, objectArray[object]);
                        break;
                    }
                }
                if (!bFound) {
                    // Did not find it, so simply insert it.
                    jsonModel.insert(index, objectArray[object]);
                }
            }
            index++;
        }
    }
```

Appendix C

Journal Papers

The journal papers are shown in chapter eight, but were also submitted to supervisor as PDFs:

- C1_JournalPaper_OriginalDraft.pdf
- C2_JournalPaper_CollaborationResult.pdf

C.1 First Revision of Abstract

Modern biomedical research often depends on compute-intensive data processing and analysis at different stages of an investigation. Galaxy(Goecks et al., 2010) is a widely used platform for genome analysis, providing a web-based interface to initiate and monitor computations executed on a server. A biomedical investigation may often involve several such computations running for a long and á priori undetermined time period. Often, projects depend on such computational results for further progress. This may lead researchers to constantly go back and forth to a computer to monitor job execution progress. Access through mobile devices could make such monitoring much more convenient and efficient. However, a web interface, such as that of Galaxy, must permit a broad range of complex analyses to be specified. These do not typically work well on the small touchscreens of mobile devices, with their relatively low precision, compared to using a mouse.

Galaxy Portal is an open source interface to the Galaxy system through smartphones and tablets. Rather than supporting the full functionality of Galaxy with its complex interface, the Galaxy Portal app limits itself to monitoring job completion, as well as inspec-

tion of results of jobs already initiated through the standard Galaxy web interface. The purpose is to provide a simple and efficient interface that can be accessed from almost anywhere at any time through a mobile device. The app provides a responsive and low-bandwidth interface with data tailored for a small screen, and supports features such as touchscreen interaction and device orientation.

Furthermore, Galaxy tracks a user's execution history in a reproducible manner, and the Galaxy Portal app provides a fast and simple way to browse the Galaxy history elements. This can be very useful as a way of quickly checking results or details of exactly how given results were produced, for instance in a discussion setting where a mobile device is more convenient to use than a computer.

Appendix D

Android Release

The Android release is available on any Android device from the Google Play store by searching for “Galaxy Portal” and typically among the top five hits, or using this link: https://play.google.com/store/apps/details?id=no.uio.bmi.claus

iOS Release

It has been approved by Apple review and is available under a separate license for iPhone and iPad through the iTunes app store: https://itunes.apple.com/us/app/galaxy-portal/id937158087

Windows and Linux Release

For Windows and Linux the binaries can be downloaded from the GitHub repository: https://github.com/Tarostar/QMLGalaxyPortal/releases

Appendix E

Collected Gartner Data

Operating System	Year	
	2007	**2008**
Symbian	63.5%	52.4%
Research in Motion	9.6%	16.6%
Microsoft Windows Mobile	12.0%	11.8%
Mac OS X	2.7%	8.2%
Linux	9.6%	8.1%
Palm OS	1.4%	1.8%
Other	1.1%	1.1%

Table E1: Gartner reported percentage share of market for smartphone sales worldwide to end users by operating systems for 2007 and 2008[83].

Operating System	Year	
	2009	**2010**
Symbian	46.9%	37.6%
Android	3.9%	22.7%
Research in Motion	19.9%	16.0%
iOS	14.4%	15.7%
Microsoft	8.7%	4.2%
Other	6.1%	3.8%

Table E2: Gartner reported percentage share of market for smartphone sales worldwide to end users by operating systems for 2009 and 2010[84].

83 See http://www.gartner.com/newsroom/id/910112; accessed April 6 2014
84 See http://www.gartner.com/newsroom/id/1543014; accessed April 6 2014

Operating System	Year	
	2012	**2013**
Android	66.4%	78.4%
iOS	19.1%	15.6%
Microsoft	2.5%	3.2%
BlackBerry	5.0%	1.9%
Other	6.9%	0.9%

Table E3: Gartner reported percentage share of market for smartphone sales worldwide to end users by operating systems for 2012 and 2013[85].

85 See http://www.gartner.com/newsroom/id/2665715; accessed April 6 2014